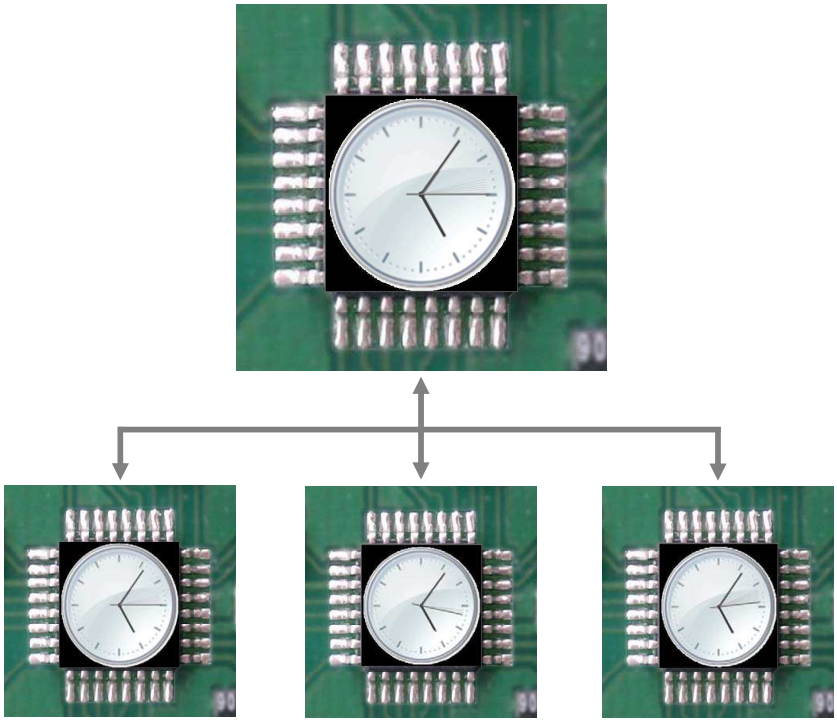
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2008

Samuel Madail

Sincronização de relógio em CAN assistida por hardware



Samuel Madail

Sincronização de relógio em CAN assistida por hardware

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica de Prof. Doutor Arnaldo Oliveira e Prof Doutor Paulo Pedreiras, Professores Auxiliares Convidados do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president

Doutor António Manuel de Brito Ferrari Almeida
Professor Catedrático da Universidade de Aveiro

vogais / examiners committee

Doutor Luis Filipe Santos Gomes
Professor Associado do Departamento de Engenharia Electrotécnica da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Doutor Arnaldo Silva Rodrigues de Oliveira
Professor Auxiliar Convidado da Universidade de Aveiro (Orientador)

Doutor Paulo Bacelar Reis Pedreiras
Professor Auxiliar Convidado da Universidade de Aveiro (Co-Orientador)

agradecimentos / acknowledgements

O trabalho realizado no âmbito desta dissertação envolveu directamente e indirectamente pessoas às quais quero agradecer. Nomeadamente:

- aos meus orientadores, Doutor Paulo Pedreiras e Doutor Arnaldo Oliveira, pois foram incansáveis no apoio que me deram ao longo deste ano, ao nível técnico e científico, fazendo tudo para que esta dissertação pudesse correr da melhor forma;

- aos meus colegas e amigos da sala 303, agora também futuros engenheiros, nomeadamente ao João Faria, ao Nelson Silva, ao Domingos Terra, ao Vítor Silva e ao Nuno Figueiredo, que apesar de não ser da nossa sala foi sempre um grande amigo. Foram todos excepcionais no apoio e nas conversas que tivemos durante a realização desta tese;

- à minha família e amigos, que sempre me apoiaram e ajudaram a ultrapassar os momentos menos bons, em particular aos meus pais Fernando e Isabel bem como à minha irmã Sandra. Por último, mas de uma forma muito especial, quero agradecer à minha namorada Sara, por todo o apoio e incentivo que me deu, pela paciência que teve comigo e pelos sacrifícios que passou para que eu pudesse realizar esta dissertação da melhor forma.

A todos eu digo um muito obrigado!

Resumo

O tempo tem sido tema de estudo desde o início da civilização até aos dias de hoje. O que mais tem mudado nesse aspecto é o rigor com que conseguimos medir o tempo. Enquanto antigamente as pessoas viviam segundo os ciclos da natureza, hoje em dia as pessoas já necessitam de utilizar relógios para se sincronizarem com outras pessoas ou procedimentos.

Este rigor é muito mais apertado em sistemas de medida e controlo presentes nas mais diversas áreas aplicacionais, tais como a automação e controlo industrial, robótica, sistemas *x-by-wire* na aviónica e indústria automóvel, entre outros. Muitas destas classes de aplicação exibem requisitos temporais estritos, requerendo um elevado rigor nos instantes de activação das tarefas desempenhadas pelo sistema bem como na medida dos instantes em que certos eventos ocorrem.

Muitos destes sistemas assentam em arquitecturas distribuídas, ou seja, são sistemas compostos por diferentes nodos "inteligentes"e autónomos que usam serviços disponibilizados por uma infra-estrutura de comunicação comum. O seu uso é hoje em dia cada vez mais frequente em detrimento das tradicionais arquitecturas centralizadas, devido ao seu custo, escalabilidade (do sistema) e em termos de partilha dos recursos do sistema. Contudo, a manutenção de um relógio comum em soluções distribuídas apresenta desafios adicionais.

Estando os diversos nodos dos sistemas distribuídos geralmente dispersos geograficamente, existe a necessidade de cada nodo do sistema possuir o seu próprio relógio local. Este facto só por si não é suficiente, dado que se não forem corrigidos, os valores dos vários relógios vão divergir uns dos outros, perdendo-se a coerência temporal do sistema. Torna-se então necessário sincronizar periodicamente os relógios para que estes se mantenham tão sincronizados quanto possível e/ou desejável.

Em consequência disso têm vindo a ser propostos diversos protocolos de sincronização de relógio especificamente concebidos para satisfazer os requisitos de sistemas deste tipo. Um destes protocolos é o IEEE 1588 (*IEEE 1588-2002 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*). A possibilidade de aplicar em diversas tecnologias de rede, a simplicidade e administração mínima, bem como os baixos requisitos em termos de recursos (CPU, memória, rede), são algumas das principais vantagens deste protocolo.

O IEEE 1588 pode ser implementado quer em hardware quer em software, sendo que o grau de precisão obtido dependente do tipo de implementação.

Nesta âmbito desta dissertação foi implementado um protocolo baseado no IEEE 1588 sobre redes CAN (*Controller Area Network*), designado por 1588Light. Esta adaptação surgiu da necessidade de simplificar o protocolo, dado que a rede CAN oferece relativamente pouca largura de banda e em muitos casos é utilizada em sistemas embutidos baseados em microcontroladores com baixa capacidade de processamento.

Primeiramente, o protocolo foi implementado ao nível de software, usando como nodos do sistema distribuído placas DETPIC com processadores PIC18F258 da Microchip. Seguidamente foi feita uma implementação do protocolo em hardware, usando para isso o controlador CAN designado por CLAN, disponibilizado na forma de núcleo de propriedade intelectual modelado em VHDL, sintetizável e implementável em FPGAs da Xilinx. A placa base de desenvolvimento da implementação em hardware foi a RC10 da Celoxica contendo uma FPGA Spartan-3.

Na implementação em software conseguiram-se precisões de algumas dezenas de microsegundos ao nível do *timer* dos microcontroladores. Ao nível da aplicação os resultados de *offset* obtidos estavam na ordem das dezenas/centenas de microsegundos. Verificou-se um bom comportamento do protocolo quando sujeito a diferentes cargas no barramento bem como pela adição de nodos *Slaves* na rede. Também se introduziu um *kernel* tempo-real, tendo-se verificado neste caso que os tempos de bloqueio impostos por este degradavam significativamente o *offset*, sendo este de algumas centenas de microsegundos.

Na implementação em hardware os resultados obtidos encontram-se na gama das unidades de microsegundo. Os valores de *offset* mantiveram-se constantes mesmo com a introdução de carga no barramento, bem como com a adição de outros nodos *Slave* na rede. Devido à possibilidade de executar múltiplas tarefas em simultâneo numa FPGA, o problema dos tempos de bloqueio na afectação do rigor do relógio não se levanta.

Conclui-se assim que para sistemas com necessidade de sincronização de relógio elevada, na casa das unidades de microsegundo, torna-se necessária um implementação assistida por hardware.

Abstract

Time has been a subject of study since the beginning of civilisation. What has changed the most in relation to this issue is the rigor with which we measure time. In the past people kept to the cycles of nature, but nowadays they need clocks to synchronise themselves with others people or processes.

This rigor is very tight in most measurement and control systems present in a range of diverse application areas such as industrial control and automation, robotics, systems x-by-wire avionics and the automotive industry, among others. Many of these application groups display strict time requirements, where a high level of accuracy is required in the activation of the tasks performed by the system as well as the measurement of instances at which certain events occur.

Many of these systems are based on distributed architectures, in other words systems composed of different "intelligent"and autonomous nodes that use services provided by a common communication infrastructure. Its use today is increasingly common in detriment of the traditional centralised architectures, due to its cost, scalability (of the system) and in terms of sharing the system's resources. However, the maintenance of a common clock in distributed solutions presents additional challenges.

As the various nodes of the distributed systems are usually spread geographically, there is a need for each node to have its own local clock. This fact alone is not enough, because if they are not corrected, the values of the various clocks will differ from each other and the time consistency of the system will be lost. It is therefore necessary to synchronise the clocks periodically in order for them to be as synchronized as possible and/or desirable.

As a result, various protocols for clock synchronisation specifically designed to meet the requirements of such systems have been proposed. One of these protocols is the IEEE 1588 (IEEE 1588-2002 Standard for the Precision Clock Synchronization Protocol for Networked Measurement and Control Systems). The possibility of applying this protocol to a range of network technologies, simplicity and minimal administration, as well as low requirements in terms of resources (CPU, memory, network), are some of the main advantages of this protocol.

The IEEE 1588 can be implemented either in hardware or in software, however, the degree of accuracy obtained is dependant upon which type of implementation is used.

This dissertation involves the implementation of a protocol based on the IEEE 1588 on CAN (Controller Area Network) networks, referred to as 1588Light. This adjustment resulted from the need to simplify the protocol, since the CAN network offers relatively little bandwidth and is often used in embedded systems or microcontrollers with low resource (ex. processing, memory) capacity.

Firstly, the protocol was implemented at the software level, using as the distributed system nodes DETPIC boards with Microchip PIC18F258 processors. Following on from that, the implementation of the protocol in hardware was carried out, with the use of the CAN controller, referred to as CLAN, and available in the form of core intellectual property modelled in VHDL, synthesisable and implementable in the Xilinx FPGAs. The development board for the implementation in hardware was the RC10 of Celoxica, containing an FPGA Spartan-3.

Through the implementation in software it was possible to obtain precision to within a few tens of microseconds at the level of the microcontroller timer. In relation to the application layer, the offset results obtained were in the order of tens/hundreds of microseconds. Protocol performance was good when subjected to different bus loads as well as the addition of slave nodes in the network. When a kernel was introduced the findings showed that the blocking times imposed by the kernel are responsible for the significant degradation of the offset, increasing it to a few hundred microseconds.

In the hardware implementation, the results were in the range of few microseconds. The offset results remained constant even with the introduction of the bus load and with the addition of other slave nodes in the network. Due to the possibility of performing multiple tasks simultaneously in a FPGA, the problem of blocking times affecting the accuracy of the clock does not arise.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Enquadramento

1.1.1 A influência do tempo e da sua medida nas nossas vidas

Não existem dúvidas que as questões relacionadas com o tempo têm sido tema de estudo desde o início da civilização até aos dias de hoje. O tempo tem sido estudado por cientistas e filósofos, tem sido uma preocupação da religião e da arte, e tem influência na nossa vida quotidiana e no nosso pensamento.

Uma das mudanças mais significativas ao longo da história neste contexto tem sido a precisão e exactidão com que somos capazes de medir o tempo. Embora não se consiga explicar o que motivou os pioneiros neste campo, é razoável assumir que sempre foram poucos os que empurraram as fronteiras do tempo apenas pelo desafio ou por ser intelectualmente interessante, pois a maioria das pessoas vivia segundo os ciclos da natureza, fossem camponeses ou pessoas da cidade.

Ao longo do último milénio a capacidade de medir o tempo tem vindo a aumentar, tendo sido alcançado melhoramentos de 15 ordens de grandeza. A maior parte dessa melhoria ocorreu nos últimos 400 anos, após a introdução do relógio de pêndulo por Huygens em meados do século XVII [BFRW00].

No nosso dia-a-dia utilizamos relógios para nos sincronizarmos com outras pessoas ou procedimentos. Quão exacto deve ser o tempo depende das circunstâncias. Por exemplo, se tivermos de apanhar um comboio, um rigor de alguns segundos ou mesmo um minuto devem ser suficientes, mas se for para sabermos quem ganhou uma corrida, um rigor na ordem dos milisegundos já poderá fazer a diferença para se saber quem é o vencedor.

1.1.2 Sistemas distribuídos

Um sistema distribuído consiste numa colecção de computadores autónomos e cooperantes ligados através de uma rede de comunicações, em que estes coordenam as suas actividades e partilham os recursos do sistema somente através da troca de mensagens. Caracterizam-se acima de tudo pela concorrência no acesso a recursos do sistema, ausência de um relógio comum bem como pela falha independente de componentes.

Muitas aplicações de tempo crítico (ex. dispositivos de medida) necessitam de relógios de tempo real com precisões na ordem dos microsegundos. Nos sistemas centralizados esta necessidade é relativamente fácil de implementar, recorrendo por exemplo, ao uso de *timers*

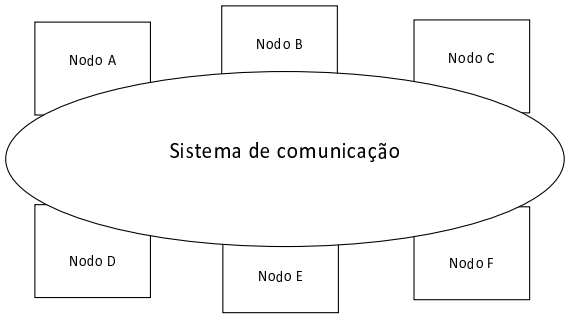


Figura 1.1: Arquitectura de um sistema distribuído.

em hardware, que podem ser directamente acedidos pelas diversas partes do sistema. Em sistemas distribuídos esta tarefa é mais complicada. Isto acontece devido à necessidade da existência de um tempo comum nos diferentes nodos do sistema. Assim pelo facto de cada periférico possuir o seu próprio relógio, torna-se necessário que todos os relógios se encontrem sincronizados com um certo rigor.

Sistemas distribuídos modernos como sistemas de medição e automação, aviões ou mesmo automóveis contêm múltiplos dispositivos de rede e frequentemente exigem o tempo rigoroso para serem capazes de sincronizar eventos, como coordenar movimentos distribuídos e correlacionar dados. Assim tornou-se necessário encontrar técnicas que permitissem manter os vários relógios sincronizados.

1.1.2.1 Sincronização de Relógio em sistemas distribuídos

> *"A man with a watch knows what time it is.*
> *A man with two watches is never sure".*
> – unknown

A citação acima referida, por um autor anónimo, revela bem a importância da existência de um tempo universal no sistema em que existem vários relógios, pois de outra forma não se saberá qual o tempo que está correcto o que em muitas situações poderá ser problemático. Isto acontece também num sistema distribuído, em que cada nodo do sistema possui o seu próprio relógio local, pelo que deve existir coerência entre os vários relógios. Um sistema distribuído para funcionar correctamente deverá garantir a coerência na precedência de acções executadas, bem como rigor no instante em que devem ser executadas essas acções por cada nodo.

1.1.3 Exemplos de aplicações que necessitam de sincronização de relógio

Skoog descreveu a sincronização de relógio como "factor chave no sucesso ou fracasso de um sistema de rede"[Sko01]. Como resultado desta condição a maioria dos computadores de rede está equipada com servidores de tempo e algoritmos de sincronização, tal como é o caso do bem conhecido *Network Time Protocol* (NTP).

A necessidade de técnicas de sincronização de relógio surge nos mais diversos domínios, como por exemplo:

. Comunicações *Time-Triggered*

Em aplicações de segurança crítica, tal como o *X-by-wire* nos carros [LA03], a variação aleatória da latência da mensagem prejudica o funcionamento dos sistemas. Em resposta a esta preocupação, os protocolos *time-triggered* estão a tornar-se soluções comuns para sistemas de segurança e tempo-real crítico. Para uma correcta operação de uma rede *time-triggered* é necessário fornecer um sistema de referência de tempo global a fim de permitir uma referência consistente na identificação do momento de activação de cada tarefa e/ou evento de comunicação.

. Sistemas de controlo

Os sistemas de controlo distribuído normalmente sofrem variações nos atrasos (*jitter*) entre a amostragem dos sensores e a reacção dos actuadores. Este *jitter* influência a estabilidade do sistema bem como muda as suas características. Para sistema de malha fechada, é imperativo que os algoritmos de controlo possuam o tempo correcto entre amostragens para cada uma das variáveis de controlo. Assim, usando relógios sincronizados o objectivo de controlo pode ser alcançado com uma menor taxa de amostragem bem como uma menor largura de banda [EC98].

. Controlo de Redundância

Em aplicações de segurança crítica, alguns graus de redundância em hardware são normalmente utilizados para satisfazer os requisitos do sistema. Usualmente, a estratégia no controlo de redundância é baseada num mecanismo de voto que necessita de um alto nível de consistência dos dados. Assim sem relógios sincronizados o mecanismo de voto poderá não funcionar correctamente, devido à não consistência dos dados.

. Aquisição síncrona de dados

A existência de uma referência de tempo global é fundamental para as exigências dos sistemas de aquisição de dados (DAQs) em ambientes distribuídos. Por exemplo, muitas DAQs usam *timestamps* para induzir a ordem total dos eventos que ocorrem em diferentes nodos. Assim se os relógios dos sistemas não estiverem sincronizados, provavelmente a ordem total dos eventos não será cumprida, levando a que os resultados obtidos não tenham uma ordem temporal correcta e por isso não possam servir em futuras comparações de causa/efeito.

Existe ainda uma vasta gama de equipamentos em que esta noção é de extrema importância, podendo referir-se aplicações na área dos sistemas militares (Figura 1.2), nomeadamente em testes de alcance, testes de voo e qualificação, sistemas operacionais, assim como na indústria de automação (Figura 1.3) no controlo de processos, robots, máquinas de empacotamento e em processos entrelaçados. Já na indústria de potência (Figura 1.4) encontra-se frequentemente a necessidade de sincronização de relógio. No controlo da geração de potência e no controlar de sistemas I/O distribuídos.

A sincronização de relógio também é de grande importância na área de testes e medição (Figura 1.5), em que para os sistemas mais exigentes é necessário uma precisão na ordem dos nanosegundos. No entanto, em muitas aplicações uma precisão entre milisegundos e microsegundos já é suficiente. Finalmente tem grande aplicabilidade na área das telecomunicações (Figura 1.6), que entre outras finalidades permite a difusão global de um relógio síncrono.

Figura 1.2: Aplicações em sistemas Militares (retirado de [Eida] e [Eid06]).

Figura 1.3: Aplicações na área da Indústria de Automação (retirado de [Eid06]).

Figura 1.4: Aplicações na área da Indústria Energética (retirado de [Eid06]).

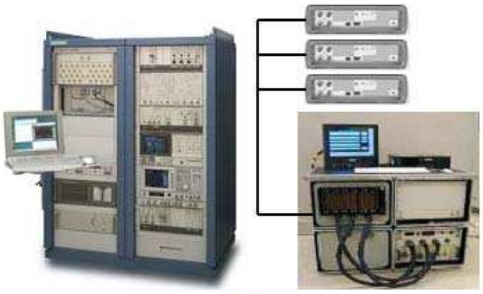


Figura 1.5: Aplicações na área de Testes&Medição (retirado de [Eid06]).

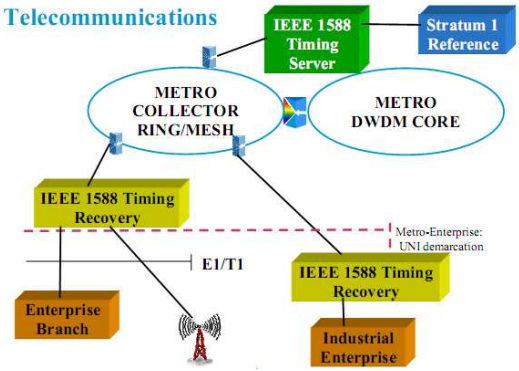


Figura 1.6: Aplicações na área das Telecomunicações (retirado de [Eid06]).

1.2 Motivação

O *Controller Area Network* (CAN) [Gmb91] é um protocolo de comunicação que continua a ter grande aplicabilidade em sistemas de automação e em ambientes industriais, sendo que as suas propriedades de tempo-real são uma mais valia para muitos sistemas. Este permite um alto nível de detecção de erros, baixos tempos de latência, flexibilidade de configuração, entre outros.

Diversas aplicações distribuídas realizadas sobre CAN requerem que todos os nodos na rede estejam sincronizados. Para isto é necessário implementar um protocolo de sincronização através deste meio de comunicação.

Sendo que já existem alguns protocolos de sincronização em CAN, tal como vem referido no Capítulo 3, nesta tese pretende-se implementar um protocolo, derivado do IEEE 1588, em CAN. O IEEE 1588 caracteriza-se sobretudo pela simplicidade e administração mínima, bem como os baixos requisitos em termos de recursos (CPU, memória, rede) exigidos, sendo que permite ser implementado em qualquer rede de comunicação, tendo assim uma clara vantagem em relação a outros protocolos de sincronização que são específicos para as redes que foram desenhados.

Todavia, algumas características do protocolo CAN, como a largura de banda limitada e o pequeno tamanho dos pacotes, colocam alguns desafios à implementação deste protocolo. Também o facto de o CAN geralmente ser aplicado a sistemas distribuídos de baixo custo, com uma capacidade de processamento ao nível dos nodos da rede limitada, implica um protocolo de sincronização simples.

1.3 Plano de trabalho

O trabalho realizado no âmbito desta dissertação passou pela familiarização com a problemática do controlo tempo real em arquitecturas distribuídas, bem como pelo estudo do protocolo de sincronização de relógio IEEE 1588. Terminada esta etapa, e estudado o protocolo de comunicação série CAN, implementou-se em software um protocolo baseado no IEEE 1588 (1588Light) para rede CAN em PICs da Microchip, nomeadamente em controladores PIC18F258.

Depois de realizada a implementação em software, passou-se à familiarização com o CLAN

disponível na forma de um núcleo de propriedade intelectual modelado em VHDL, sintetizável e implementável em FPGAs da Xilinx. O CLAN é um controlador CAN feito de raiz no Departamento de Electrónica, Telecomunicações e Informática na Universidade de Aveiro. Este será importante na implementação do 1588Light em FPGAs da Xilinx, através do uso da placa Spartan-3 da Celoxica.

Por fim, após tirar os resultados das implementações realizadas, procedeu-se à análise do desempenho das mesmas.

1.4 Organização da tese

Neste capítulo é feita uma introdução ao trabalho, contextualizando a dissertação e a motivação para a realização da mesma. Para dissertar sobre o tema proposto, esta tese para além desta introdução, encontra-se estruturada do seguinte modo:

. Sincronização de Relógio em Sistemas Distribuídos

Neste capítulo são abordados os conceitos gerais relativos aos sistemas distribuídos, bem como a necessidade de em muitos casos estes necessitarem de um visão global do tempo, e por isso necessitarem de técnicas que lhes permitam terem os relógios dos vários controladores sincronizados. São também referidas as várias técnicas de *timestamp* que se podem utilizar e quais os seus prós e contras. Finalmente são abordados os principais protocolos de sincronização de relógio em sistemas distribuídos, nomeadamente o NTP, o IEEE 1588 e o SynUTC, sendo que o IEEE 1588, por ser também tema desta dissertação, terá uma abordagem mais profunda.

. Sincronização de relógio em CAN

Este capítulo começa por abordar o protocolo de comunicação CAN, bem como refere algumas das suas características que se revelam úteis na sincronização de relógio. Dado não existir ainda uma implementação do IEEE 1588 em CAN, este capítulo também inclui a adaptação proposta do IEEE 1588 para *DeviceNet*, bem como as principais técnicas de sincronização de relógio em CAN encontradas.

. Implementação em Software

Neste capítulo é apresentada a implementação em software de um protocolo baseado no IEEE 1588 (1588Light) sobre a rede CAN. Será feita uma abordagem ao protocolo implementado e às suas particularidades. Posteriormente será descrita a implementação propriamente dita, sendo explicada a sua arquitectura e implementação de forma detalhada, bem como as opções tomadas e o motivo pelas quais foram tomadas.

. Implementação em Hardware

Este capítulo apresenta a implementação do 1588Light realizada em hardware. Assim será descrita a arquitectura global do sistema bem como explicada a implementação de cada bloco. Serão ainda referidos os recursos utilizados, frequência de funcionamento da aplicação e placa de desenvolvimento utilizada.

. Resultados

Este capítulo encontra-se dividido em três partes distintas. Primeiramente serão brevemente apresentadas as principais ferramentas desenvolvidas, que permitiram retirar

os resultados relativos às implementações do protocolo. Em seguida são apresentados os resultados relativos à implementação em software realizados em diferentes condições. Por fim, são também apresentados os resultados da implementação em hardware bem como é feita uma análise comparativa entre os resultados das duas implementações.

. Conclusão

Por último, neste capítulo são apresentadas as conclusões desta dissertação. Será feito um sumário ao trabalho realizado bem como comentados e explicados os resultados obtidos. São sugeridos também possíveis melhoramentos no protocolo 1588Light bem como nas implementações realizadas.

Capítulo 2

Sincronização de relógio em sistemas distribuídos

2.1 Introdução

Os sistemas distribuídos são cada vez mais utilizados e objecto de estudo. Kopetz [CDK94] classifica-os como sendo uma colecção de dispositivos ligados por uma rede de comunicação e equipados com um software de sistemas distribuídos. Este software permite que os vários dispositivos coordenem as suas actividades e partilhem os recursos do sistema, sejam eles hardware, software ou dados. Os seus utilizadores devem percebe-lo como um único sistema, ainda que este possa estar implementado em vários computadores localizados em sítios diferentes.

Os sistemas distribuídos bem concebidos, possuem características que os tornam bastante interessantes e robustos levando a que cada vez mais se opte por soluções distribuídas. Por exemplo, a escalabilidade do sistema, permite que o sistema possa crescer sem limite a esse desenvolvimento. Também a dependabilidade, dado que garante a manutenção das características do sistema mesmo na presença de falhas ou erros seja de que nível for, sendo conseguido através da criação de zonas estanque que evitam a propagação de erros ou falhas pelo resto do sistema. Outra característica neste tipo de sistemas é a composabilidade em relação a uma propriedade, garantindo que essa propriedade não deixará te ter efeito pela integração do sistema, desde que tenha sido garantida ao nível do sub-sistema.

É relativamente fácil de perceber porque estes sistemas são tão populares. Estes permitem a partilha de informação e recursos sobre uma grande extensão geográfica e são usualmente melhores do que os tradicionais sistemas centralizados em termos de partilha de recursos, custo global, expansão e autonomia do sistema.

> *"You know you have one when the crash of a computer you've never heard of stops you from getting any work done".*

Mas nem tudo são vantagens. Na citação anterior, Leslie Lamport [Mul94] refere a existência de insuficiências e fragilidades com as implementações baseadas em sistemas distribuídos. Em virtude da natureza distribuída, estes sistemas têm de lidar com comunicações incertas e inseguras e falhas independentes. Estes problemas tornam-se graves quando o sistema opera em aplicações de tempo real crítico, tais como, sistemas aeronáuticos, sistemas

de suporte à vida, centrais nucleares, sistemas *drive-by-wire* e sistemas de fabrico de componentes integrados. Comum a todas estas aplicações está a exigência de máxima fiabilidade e alta performance dos dispositivos controladores, visto que uma simples falha do controlador numa destas aplicações pode levar a uma situação de desastre.

Mesmo quando inicialmente definido com precisão, o relógio real será diferente depois de algum tempo devido ao *drift* do relógio, causado pela contagem de tempo dos relógios com taxas ligeiramente diferentes. Estes *drifts* do relógio podem variar ao longo do tempo, variam com a temperatura e chegam mesmo a variar com o envelhecimento dos componentes físicos usados para a sua realização.

Em sistemas centralizados a variação do relógio do sistema não causa inconsistências. Nestes, o servidor centralizado dita o tempo do sistema, sendo que quando algum processo necessita de saber o tempo, basta aceder ao tempo do servidor centralizado. Logo, se o processo A pergunta o tempo, e pouco depois o processo B também lê o tempo, o valor que B obtém irá ser certamente igual ou superior ao valor obtido em A [TvS02].

Com o crescente número de aplicações distribuídas, tais como, aplicações de controlo de processos, aplicações de transformação de processos, ou protocolos de comunicação, levou os dispositivos autónomos a necessitarem de cooperação para a iniciação de acções ou guarda de eventos. Portanto, a ordenação causal é frequentemente exigida, o que leva os vários nodos a necessitarem de ter os seus relógios sincronizados para que todos tenham aproximadamente a mesma visão do tempo. Assim, quando os relógios sincronizados estão disponíveis, a performance dos sistemas distribuídos pode ser melhorada pela redução da comunicação, como se pode ver em [Kis93] para algumas utilizações práticas neste tipo de sistemas.

Neste processo os relógios são verificados periodicamente e são ajustados se necessário. Isto é feito principalmente pela comunicação entre os melhores e os piores relógios. Relógios imprecisos e relógios que necessitem de grande rigor têm de ser corrigidos com mais frequência.

Neste capítulo serão apresentados os principais protocolos existentes para a sincronização de sistemas distribuídos, tais como o NTP, IEEE 1588 e o SynUTC, sendo que o IEEE 1588 merecerá um maior destaque. Inicialmente serão referidas ainda as técnicas existentes para a efectuação de *timestamp*, referindo as fontes de erro associadas a cada uma, bem como as suas vantagens e desvantagens.

2.2 *Timestamping*

No decorrer dos protocolos de comunicação frequentemente é necessário efectuar o *timestamp* às mensagens trocadas entre os vários nodos. Este consiste em registar o instante temporal em que são enviadas ou recebidas as mensagens de sincronização.

O rigor do *timestamping* é ponto fulcral na qualidade de sincronização de relógio, visto que a incerteza dos *timestamps* levará a uma imprecisão nos cálculos para o ajuste do relógio. O rigor deste depende sobretudo da precisão dos relógios bem como do local em que é feito. O primeiro aspecto resolve-se com a utilização de relógios com maior precisão, o segundo aspecto é solucionado efectuando o *timestamp* o mais perto possível do meio de comunicação, por forma a minimizar as fontes de interferência.

Existem três possibilidades diferentes de realizar o *timestamp*, tal como pode ser visto na Figura 2.1. São eles *timestamping* ao nível da aplicação, ao nível do *device-driver* e ao nível do hardware (visto de cima para baixo).

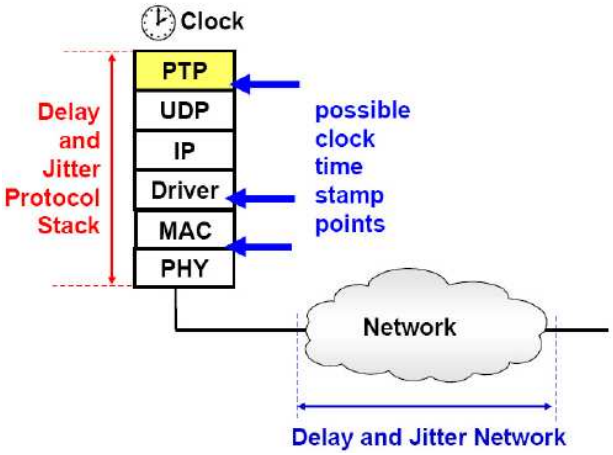


Figura 2.1: Possibilidades de *timestamping* do tempo de relógio na stack do protocolo (retirado de [WB]).

2.2.1 *Timestamp* assistido por hardware

Na abordagem assistida por hardware, o *timestamping* é efectuado por um interface independente, entre a camada física (PHY) e a camada MAC, o mais próximo possível do meio de comunicação. Esta é necessária para muitas aplicações industriais que precisam de algumas ordens de magnitude acima das conseguidas com as soluções em software (tipicamente na ordem dos nanosegundos). Este interface intermédio é geralmente uma FPGA que está dedicada a lidar com todas as acções de tempo crítico.

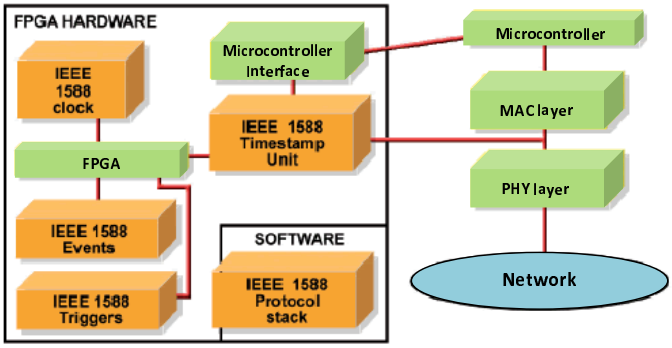


Figura 2.2: Diagrama de blocos da implementação do IEEE 1588 numa FPGA (baseado em [Tan05]).

Dada a flexibilidade inerente à FPGA, as implementações variam, mas geralmente a FPGA fica situada fora do normal caminho de comunicação. Com a passagem de pacotes entre a camada física e MAC são registados os seus tempos (através do *timestamping*). Para melhorar os resultados deve-se ter em conta a escolha da FPGA, a frequência a que a FPGA trabalha para as operações de *timestamping* e detalhes específicos da implementação. Contudo, esta abordagem exige mais esforços por parte do desenhador do sistema, uma maior complexidade

de todas as funções do protocolo de sincronização que têm que ser desenvolvidas, levando também a um custo adicional de componentes (ex. FPGA). Este tipo de implementação pode ser visto na Figura 2.2.

2.2.2 *Timestamp* em software

No *timestamping* em software as funções de sincronização são tratadas como software adicional. A Figura 2.3 ilustra a divisão dessas funções num sistema baseado em software. Nestes sistemas a incerteza associada aos instantes de execução das tarefas e de atendimento das interrupções, causa atrasos não determinísticos nos eventos de timestamping relativos à sincronização de relógio, que irão assim impor limites para a precisão que se consegue obter.

Dentro do *timestamp* em software pode-se considerar dois subtipos, nomeadamente o *timestamp* ao nível do *device-driver* e ao nível da aplicação.

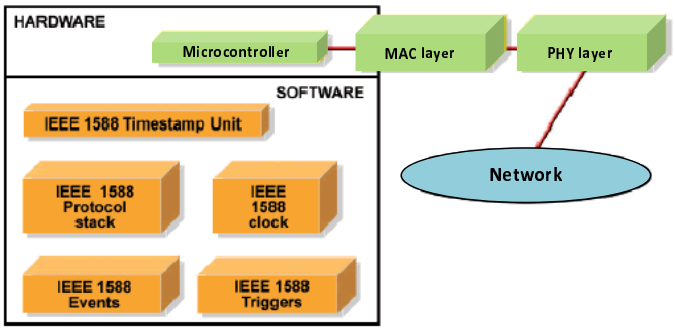


Figura 2.3: Diagrama de blocos da implementação do IEEE 1588 em software (baseado em [Tan05]).

2.2.2.1 *Timestamp* ao nível do *device-driver*

Uma possibilidade de *timestamping* em software é actuando ao nível do *device-driver*. Esta elimina as grandes variações de atraso da passagem das mensagens pela *stack* do protocolo. Este *jitter* depende do tipo de sistema operacional, do conjunto de aplicações que correm no sistema, do sistema de hardware, do sistema de interrupções e de outros factores.

O *timestamp* ao nível do *device-driver* é uma óptima solução em software mas requer a alteração do *driver* de rede. Estas alterações, permitem que o *timestamp* das mensagens recebidas seja feito logo no início da rotina de serviço à interrupção (ISR) servindo a interface de rede. Além disso, nas mensagens transmitidas é feito o *timestamping* no final da rotina de envio, ou seja, quando a mensagem passa para o hardware (controlador MAC).

Com esta técnica, tipicamente é possível obter precisões na ordem de algumas centenas de microsegundos.

2.2.2.2 *Timestamp* ao nível da aplicação

Também existe a possibilidade de efectuar o *timestamp* ao nível da aplicação. Aqui além do *jitter* que é introduzido numa abordagem ao nível do *device-driver*, são também introduzidos atrasos, em geral não determinísticos, devido ao tempo de espera a que a tarefa associada ao

timestamping é sujeita. Este geralmente corresponde ao tempo decorrido desde que a tarefa fica pronta a executar até ser realmente executada.

Depende do algoritmo de escalonamento e dos atributos utilizados (ex. prioridade relativa) sendo tipicamente os valores muito superiores aos encontrados ao nível do *device-driver*.

Assim em sistemas baseados em software ao nível da aplicação, normalmente não se conseguem precisões ponto-a-ponto inferiores a alguns milisegundos.

2.2.3 Análise comparativa

Em suma, existem impactos significativos na facilidade de desenvolvimento e na precisão de sincronização de relógio conseguida dependendo das escolhas que são feitas ao implementar o protocolo de sincronização. Uma escolha adequada da arquitectura que suporte o nível de sincronização de relógio necessário irá permitir ao desenhador desenvolver produtos industriais com performance suficiente, sem o aumento substancial do esforço no desenho e no custo.

Mediante as expectativas de sincronização e a aplicação em que se quer implementar o protocolo deve-se escolher o método correcto, ou seja, o método que permite obter o sincronismo com valores de precisão adequados. Na Tabela 2.4 é ilustrado um guia que poderá ajudar nessa escolha.

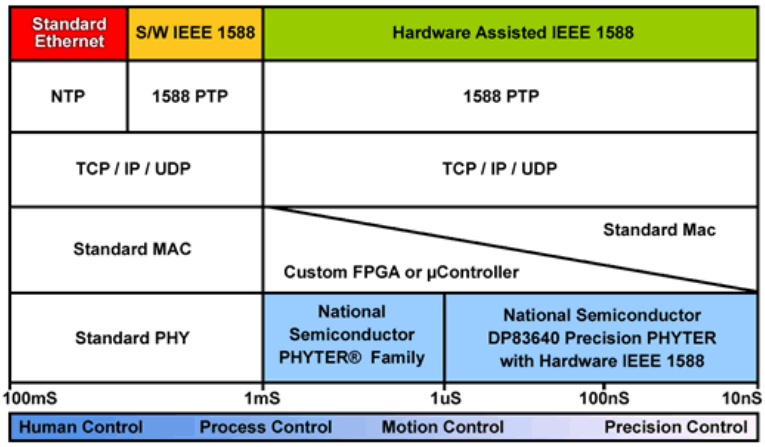


Figura 2.4: Expectativas de sincronização baseadas nas escolhas de arquitectura (retirado de [Tan05]).

2.3 Fontes de incerteza num sistema de sincronização

Para além das questões associadas ao *timestamping*, existem outros factores que afectam a incerteza do relógio. Na Tabela 2.1 são sucintamente apresentadas as principais fontes de erro de um sistema de sincronização, bem como o seu impacto na sincronização de relógio e como podem ser melhoradas.

Fonte de incerteza	**Impacto na sincronização do relógio**	**Potencial melhoramento**
Timestamp	A incerteza dos *timestamps* reduz a precisão dos cálculos fundamentais para o ajuste do relógio.	Mover a unidade de *timestamp* para mais próximo do meio de comunicação reduz a incerteza. A resolução fina dos relógios melhora a precisão do *timestamp*. Desenhos determinísticos fornecem ambientes limpos para obter o *timestamp*.
Mudança da frequência de *offset* relativa para o *Master*	A frequência de *offset* entre o *Master* e o *Slave* é corrigida através dos cálculos do PTP. Como o *drift* ocorre, os ajustes do relógio local podem temporariamente reduzir a sincronização entre os relógios.	Um ajuste granular fino pode reduzir a incerteza no relógio local.
Estabilidade da frequência no relógio local	A sincronização do relógio local para o relógio *Master* é baseada na qualidade da referência do relógio. Consegue-se, com uma melhor qualidade do relógio de referência, um menor *drift*.	Utilização de osciladores com baixo *drift* por alteração da temperatura.
A incerteza da latência no atraso dos pacotes na rede	Latências não determinísticas que estão introduzidas em algumas partes da rede irão aumentar o erro no tempo de compensação.	Componentes de rede próprios e a selecção da topologia de rede reduzem a latência.
Assimetria no caminho dos dados reduz a precisão do protocolo IEEE 1588	O protocolo PTP assume a simetria entre o caminho de transmissão e recepção nos seus cálculos.	Desenho da rede e escolha de cabos são necessárias para maximizar a simetria entre o caminho de transmissão e recepção.
Atraso na manutenção dos pacotes do IEEE 1588.	Longos atrasos levam a tentativa de sincronizar a falhar. A manutenção dos atrasos que afectam a exactidão reduzem a precisão do sistema de sincronização.	Reduzir a carga associada com as operações de sincronização do relógio para melhorar a resposta aos pacotes PTP.

Tabela 2.1: Fontes de erro do sistema de sincronização

2.4 NTP

O NTP (*Network Time Protocol*) é o protocolo usado mais frequentemente na sincronização do tempo na Internet. Este protocolo é usado para conseguir a sincronização de relógio entre o tempo do servidor e os seus clientes. Numa rede local sem demasiado equipamento de rede, tal como *switches* e *routers* podem-se conseguir precisões de 10 milisegundos.

Um levantamento maciço do protocolo na Internet global revelou que a maioria dos relógios NTP estavam dentro de 21 ms da sua fonte de sincronismo e todos estavam dentro de 29 ms, em média.[DLM02]

A arquitectura do NTP usa o conceito de camada, num modelo hierárquico (tipo árvore), em que os servidores fornecem o tempo para os níveis inferiores (clientes). Primeiro, os servidores são ajustados pela raiz da árvore, que geralmente são fontes externas de relógio (ex. GPS), que servirão como referência temporal do sistema (camada 0). É permitido um máximo de 15 camadas.

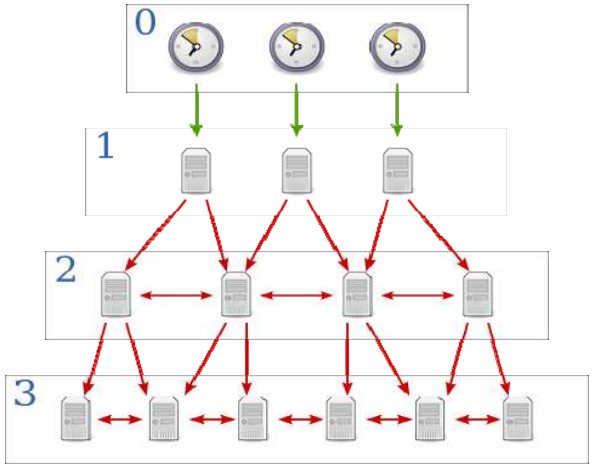


Figura 2.5: Modelo hierarquico utilizado no NTP. As linhas verdes representam ligações directas e as linhas vermelhas indicam ligações de rede (retirada de [Esh07]).

O princípio básico da sincronização no NTP é que cada cliente envie periodicamente um pedido para os servidores para que estes respondam com o seu *timestamp* local. A lista de servidores desejáveis é mantida em cada cliente e é adaptada periodicamente. O algoritmo interno avalia os *timestamps* de todos os servidores para seleccionar o melhor, com a menor camada e distância de sincronização. Este *timestamp* será usado para ajustar o relógio do cliente. O tempo de *offset* é calculado de uma colecção de quatro *timestamps* (2 do servidor e 2 dele próprio), tal como ilustrado na Figura 2.6.

O cliente, envia uma mensagem *NTP_request* que contém a origem do tempo CT1 (*timestamp* do cliente). Ao receber o *NTP_request*, o servidor gera um *timestamp* da recepção da mensagem ST1 (*timestamp* do servidor). Depois de processado o pedido, o servidor envia de volta para o cliente, a mensagem *NTP_response* com o timestamp ST2. O cliente recebe o *NTP_response* e gera o *timestamp* CT2. Seguidamente o cliente efectua o cálculo com os *timestamps* obtidos de forma a ajustar o seu relógio local.

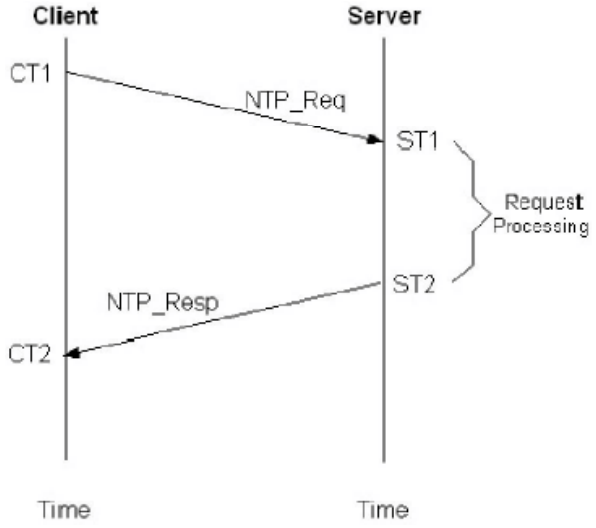


Figura 2.6: Mensagens trocadas durante a sincronização por NTP (retirada de [DLM02]).

2.5 SynUTC

O SynUTC (*Synchronized Universal Time Coordinated*) [HGH+02] é um protocolo de sincronização que permite a difusão da alta precisão do tempo de GPS através dos nodos colocados na rede Ethernet.

Através da troca de pacotes em conjugação com algumas mudanças no suporte a hardware que têm de ser efectuadas nos nodos e *switches*, podem-se conseguir precisões na ordem dos 100 ns. As aplicações podem utilizar este tempo de alta resolução tanto para efectuar *timestamps*, como para gerar eventos ao nível de software e de hardware, o que representa uma mais-valia.

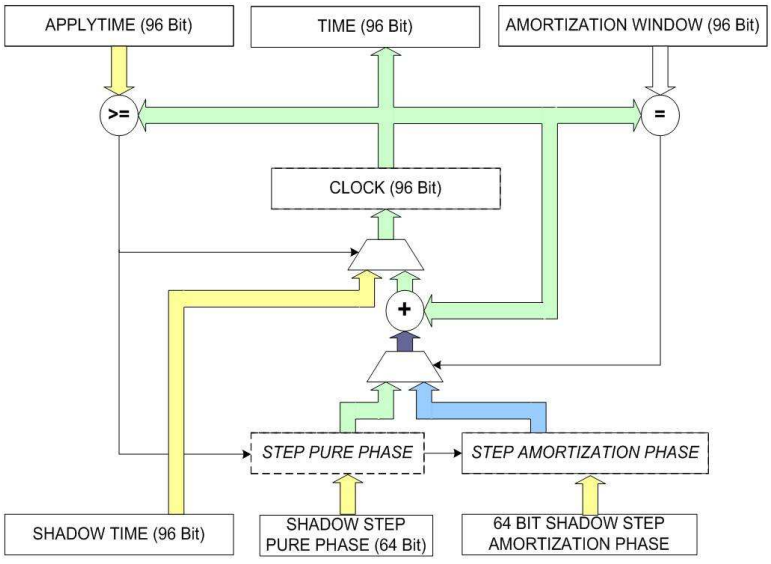


Figura 2.7: Estrutura de um *adder-based clock* (retirada de [HGH+02]).

O SynUTC baseia-se na inserção de informação de tempo altamente precisa em pacotes de dados na MII (*Media Independent Interface*), situada entre a camada física e o controlador de rede, mediante a transmissão e recepção. Como consequência, cada nodo tem acesso a informação precisa (*timestamp*) do envio e recepção de pacotes, sendo necessário equipá-lo com uma NIC (*Network Interface Card*) específica. Este *chip* contém um *adder-based clock* ajustável e registos de *timestamp* bem como um microcontrolador a executar o algoritmo de sincronização.

Basicamente, um *adder-based clock* é um relógio que usa um componente de alta precisão em vez de um simples contador que soma o tempo decorrido entre *ticks*. Assim consegue-se uma taxa de ajuste de frequência muito fina (na ordem dos ns/s) e suporte para um ajuste via amortização contínua ao nível do hardware. Na Figura 2.7 é possível ver a estrutura de um *adder-based clock*.

Esta tecnologia é bastante genérica, dado que o suporte de hardware pode ser usado com qualquer NIC baseado no MII. Além disso, todo o processo de sincronização excepto o serviço de troca de mensagens de sincronização pode ser feito no NIC. Isto permite a grande vantagem de não limitar o SynUTC à rede Ethernet, podendo assim ser aplicado em qualquer rede de dados orientada a pacotes. Depois da avaliação com sucesso do protótipo desenvolvido, tem-se feito demonstrações para facilitar a transferência do SynUTC para aplicações comerciais.

2.6 IEEE 1588

O protocolo IEEE 1588 também designado por *Precision Time Protocol* (PTP), fornece um método padrão para sincronizar dispositivos em redes, podendo em algumas situações ser obtida uma precisão dentro dos microsegundos. A exactidão pode estar na gama dos nanosegundos se este protocolo usar *timestamps* ao nível de hardware (ex. FPGA), embora este protocolo não imponha qualquer tipo de exigência de qualidade.

Neste protocolo depois de escolhido o relógio *Master* da rede (ou sub-rede), os relógios *Slave* sincronizam com o relógio *Master*, assegurando que os eventos em todos os dispositivos têm o mesmo tempo base. Este protocolo está optimizado para sistemas distribuídos, utilizando o mínimo de largura de banda da rede e apresentando baixas cargas de processamento.

O PTP necessita de uma administração muito reduzida, permitindo também a implementação do protocolo com dispositivos baratos e sem grande qualidade. Além disso, o protocolo está disponível para uma vasta gama de tecnologias de rede, sendo por isso passível de ser implementado em diferentes redes de comunicação.

2.6.1 Vista geral sobre o protocolo IEEE 1588

O IEEE 1588 permite fazer a sincronização dos relógios dentro da rede, através de um processo em que são necessários efectuar dois passos para se conseguir a sincronização dos dispositivos:

1^{o}passo - Os relógios são organizados numa hierarquia *Master-Slave* (baseada na observação da informação das propriedades dos relógios). Assim é determinado se os dispositivos da rede irão funcionar como relógio *Master* ou *Slave*.

2^{o}passo - Cada *Slave* sincroniza com o seu *Master* (baseado na troca de mensagens entre o relógio *Slave* e o seu *Master*). Neste processo é medido e corrigido o desvio causado pelo *offset* do relógio e atraso na rede.

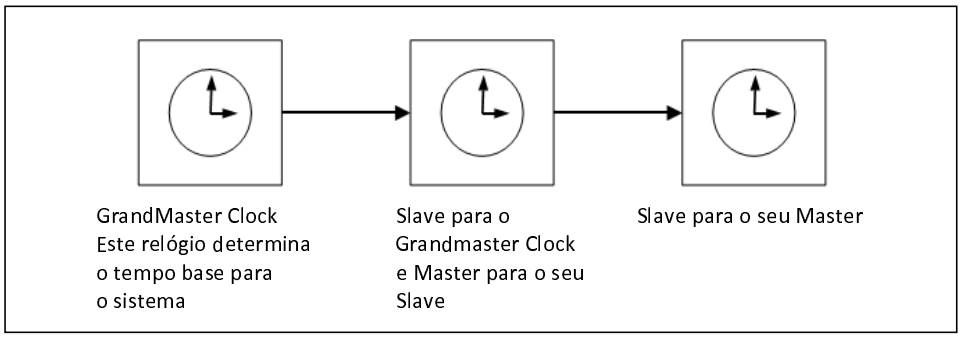


Figura 2.8: Organização hierárquica dos relógios numa rede em que o protocolo IEEE 1588 é aplicado.

Num sistema simples (só um caminho na rede) quando inicializado este protocolo utiliza o algoritmo *Best Master Clock* para saber qual o relógio mais preciso. O relógio com melhores características irá tornar-se no relógio *Master* e todos os outros passaram a ser relógios *Slave*.

Devido à diferença entre os relógios (*Master* e *Slave*) ser uma combinação de dois atrasos, ou seja, o *offset* entre os relógios e o atraso relativo ao envio da mensagem, este processo terá que ser feito em duas etapas: a correcção do *offset* e correcção do atraso. Estas duas etapas encontram-se ilustradas na Figura 2.9.

Correcção do *offset*: O relógio *Master* envia uma mensagem *Sync* e posteriormente uma mensagem *Follow-up*. Na primeira o *Slave* utiliza o seu relógio local para marcar o tempo de chegada da mensagem (82s), na segunda mensagem o *Master* envia a altura em que foi enviada a mensagem *Sync* segundo o seu relógio local (100s). Com estas informações o *Slave* será capaz de calcular a diferença entre as duas marcas temporais que representa o *offset* do *Slave* mais o atraso de transmissão (100-82=18s). Logo a seguir o relógio *Slave* ajusta o seu relógio local através da diferença obtida anteriormente (83+18=101s).

Correcção do atraso: é feito novo envio das mensagens *Sync* e *Follow-up* para calcular o atraso do *Master* para o *Slave* (105-105=0s). Esta etapa apenas é necessária se houver variações de atraso da rede. Em seguida o *Slave* envia uma mensagem *Delay Request* e regista o momento de envio (108s). O *Master* regista o momento da chegada da mensagem e envia-o numa mensagem *Delay Response* (112s). Assim o *Slave* consegue calcular a diferença entre as marcas temporais que representam o atraso do *Slave* para o *Master* (112-108=4s). O *Slave* faz então o cálculo da média relativo ao atraso nos dois sentidos e ajusta o relógio ((0+4)/2=2s).

Dado que os relógios funcionam independentemente, periodicamente tem que se repetir todo o processo, corrigindo o *offset* e o atraso para que assim os relógios permaneçam sempre sincronizados. Estes períodos para se repetir o processo assumem normalmente um destes valores predefinidos: 1, 2, 8, 16 ou 64 segundos [LA03].

2.6.1.1 Mensagens trocadas entre um relógio *Master* e relógio *Slave*

São cinco os tipos de mensagem entre os relógios que participam no protocolo PTP.

Mensagens *Sync*: são usadas por relógios no estado *Master*, contendo informação sobre a caracterização do relógio e uma estimativa do tempo de envio desta mensagem(t1), quando

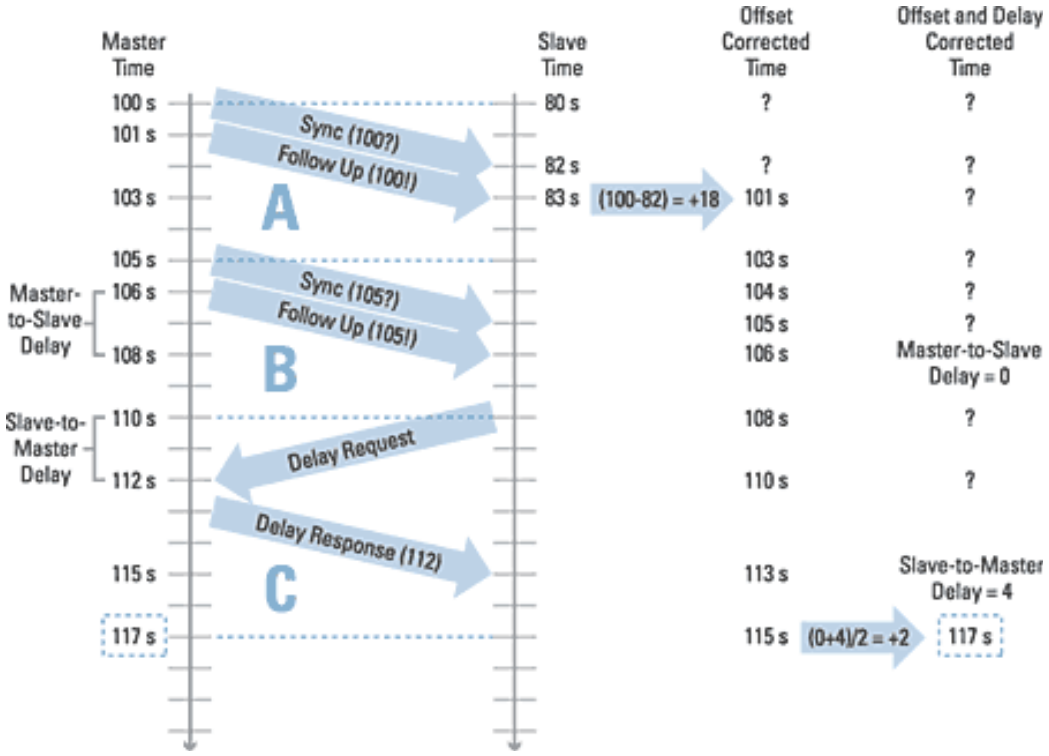


Figura 2.9: Exemplo simplificado do IEEE 1588 que mostra a correcção do offset e do atraso para sincronizar os relógios (retirado de [McC07]).

esta mensagem é recebida pelo relógio *Slave* é marcado esse tempo (t2). Para melhor exactidão estas mensagens devem ser identificadas e detectadas o mais perto possível da camada física para se obter um tempo de envio (ou recepção) mais preciso.

Mensagens *Follow_Up*: são usadas por relógios no estado *Master*, tendo sempre associado como precedente a mensagem *Sync*. Estas mensagens contêm o tempo preciso de envio da mensagem *Sync* (=t1) que deve ser medido tão perto quanto possível da camada física da rede. Quando esta mensagem é recebida pelo relógio *Slave* o tempo preciso de envio (=t1 que esta contido na mensagem) é utilizado nos cálculos para a correcção *offset* juntamente com o preciso tempo de chegada (t2).

Mensagens *Delay_Req*: são usadas por relógios no estado *Slave*. O *Slave* mede e guarda o tempo em que foi enviada a mensagem (t3). Quando esta chega ao relógio *Master* é registado o tempo de recepção (t4). Para melhor exactidão estas mensagens devem ser identificadas e detectadas o mais perto possível da camada física permitindo um obter um tempo de envio (ou recepção) mais preciso.

Mensagens *Delay_Resp*: são usadas por relógios no estado *Master*, tendo sempre associado como precedente a mensagem *Delay_Req* vinda do relógio *Slave*. Estas mensagens contêm o tempo associado à chegada da mensagem *Delay_Req* ao relógio *Master* (=t4). Quando esta mensagem chega ao relógio *Slave* o tempo associado à recepção da mensagem (=t4 que esta contido na mensagem) é utilizado nos cálculos para a correcção do atraso juntamente com o tempo de envio da mensagem *Delay_Req* (t3).

Mensagens *Management*: estas mensagens fornecem uma visibilidade externa de dados do sistema mantidas dentro de cada relógio. Elas suportam um mecanismo que permite modificar certos parâmetros nos dados do sistema, tais como, *sync_interval*, *subdomain_name*, entre outros. Estas mensagens também proporcionam um mecanismo para efectuar determinadas mudanças de estado como por exemplo, inicialização, desactivação, mudança do relógio *GrandMaster*.

2.6.2 Performance

A maioria das implementações do PTP conseguem um rigor de microsegundos, mas a performance é altamente dependente de como o sistema é implementado. No protocolo IEEE 1588 existem factores importantes que influenciam a performance do sistema, nomeadamente a frequência e estabilidade dos relógios, a topologia da rede e as variações de carga no barramento.

O *timestamping* é outro factor que afecta o rigor do relógio, nomeadamente na forma como é capturado. Se este for impreciso, os cálculos e consequentemente o ajuste no relógio também o será. Neste contexto o protocolo IEEE 1588 prevê que a captura de *timestamp* possa ser feita em alto nível (software) ou em baixo nível (hardware).

Assim, mediante os resultados de precisão e exactidão exigidos pelo sistema deve-se escolher a melhor forma de implementar o protocolo. Em situações que seja necessário valores de relógio altamente precisos, tem de se enveredar por soluções em hardware, o que por si só torna o sistema significativamente mais caro. Esta maior precisão, conseguida através de uma solução em hardware, advém da precisão com que se consegue obter os *timestamps* no decorrer do protocolo, que permitem assim evitar grande parte do *jitter* introduzido nos *timestamps* das mensagens quando estes são feitos em software. Contudo uma solução em software torna-se significativamente mais barata e, para muitas aplicações, precisões de algumas centenas de microsegundos são suficientes.

Na tabela 2.2 é dada uma ideia dos valores aproximados para o *jitter* introduzidos no *timestamp* mediante os diferentes níveis.

Local em que é feito o *timestamp* da mensagem de sincronização	**Gama de *jitter* aproximado**
No nível da aplicação de software	500μseg até 5mseg
No *kernel* do sistema operativo	10μseg até 100mseg
No hardware do controlador de comunicação	menos de 10μseg

Tabela 2.2: *Jitter* aproximado da mensagem de sincronização.

Nas soluções baseadas somente em software, além de todo o *jitter* introduzido no *timestamp* desde a camada física até ao alto nível da aplicação PTP, estas soluções também poderão representar um significativo aumento na carga de processamento. Este aumento de carga poderá prejudicar as outras aplicações a decorrem no nodo, bem como estas prejudicarem a aplicação PTP na obtenção dos *timestamps* devido aos maiores tempos de bloqueio que são introduzidos. Este problema torna-se realmente importante quando em vez de CPUs dedicados ao PTP ou com grande capacidade de processamento, temos sistemas embutidos ou microcontroladores, dado que estes se apresentam bastante limitados na capacidade de processamento.

2.6.3 Condições gerais do sistema

O IEEE 1588 sincroniza relógios de tempo-real em modelos distribuídos, criando uma noção comum do tempo em todo o sistema. Do ponto de vista do sistema e segundo [Eid05c] estas características são importantes:

. Precisão

. Exactidão

. Época

. Dependência da topologia da rede

. Recursos necessários

. Comportamento dinâmico

2.6.3.1 Precisão:

Existem duas facetas da precisão na comum noção do tempo: a precisão temporal entre os módulos individuais, e a precisão temporal no conjunto dos módulos.

A precisão temporal num módulo individual é medida pela dispersão das medidas temporais feitas no módulo relativamente ao módulo de tempo base. Tipicamente relógios de tempo-real serão implementados em hardware através de um contador conduzido por um oscilador local. A precisão deste será função da resolução do contador e das características de ruído do oscilador.

A precisão temporal de um grupo de módulos depende da precisão temporal de cada módulo individual assim como da precisão que o IEEE 1588 é capaz de conseguir na sincronização dos relógios dentro do sistema. Isto é ilustrado na Figura 2.10.

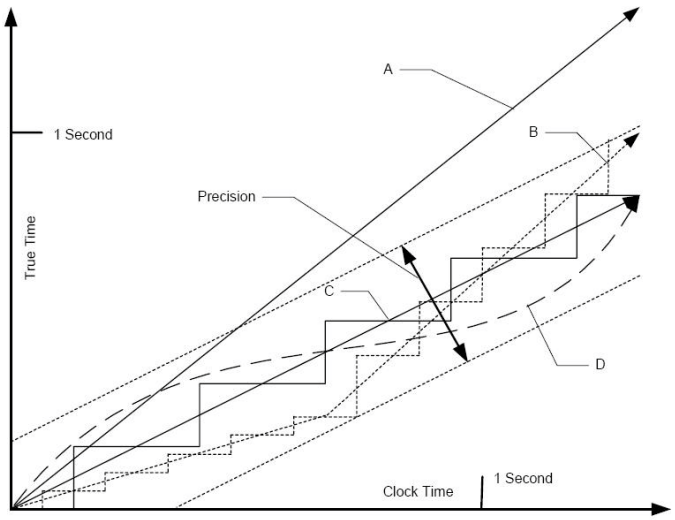


Figura 2.10: Dispersão temporal de alguns relógios em relação ao tempo base (retirado de [Eid05c])

Esta figura mostra a dispersão temporal de cada um dos três relógios B, C, D relativamente ao tempo verdadeiro (*True Time*) A. Cada um dos relógios exibe diferentes resoluções e *jitters*. O *jitter* surge de variações aleatórias na frequência do oscilador que causam variações na contagem deste em pequenas escalas numa base aleatória. Por exemplo, o relógio D tem uma resolução muito fina (não visível neste desenho) mas exibe um *jitter* que causa uma curva ondulada acima e abaixo do seu valor médio. O relógio C tem uma resolução muito grosseira mas contém pouco *jitter* em relação ao seu valor médio. O relógio B tem melhor resolução mas tem um *jitter* significativo. A posição relativa da dispersão de relógios individuais mudará uns em relação aos outros devido ao normal funcionamento do IEEE 1588, que opera para manter a média dos tempos de todos os *Slaves* idêntica à do relógio *Master*.

A dimensão desta dispersão define a largura de dispersão do sistema ou a precisão de tempo base. Portanto os relógios B, C, e D têm uma largura de precisão do sistema indicada pelas duas linhas ponteadas na Figura 2.10. O resultado efectivo disto é que quando comparando os tempos medidos em dois relógios diferentes do sistema a largura de precisão limita tanto os cálculos aritméticos como relacionais.

2.6.3.2 Exactidão

O significado comum de exactidão é concordância entre a medida realizada do segundo no sistema e a definição internacional do segundo. Por exemplo, na Figura 2.10 os relógios B, C, e D têm o mesmo tempo base desde que estejam sincronizados, mas o segundo neste tempo base é claramente menor do que o segundo no tempo base do relógio A. Já que o IEEE 1588 é um protocolo de sincronização *Master-Slave*, a exactidão é determinada pela exactidão do tempo base do relógio GrandMaster que é a raiz da hierarquia do relógio estabelecida pelo protocolo. Todos os relógios ajustam os seus tempos base com o GrandMaster ficando com a mesma exactidão que este.

2.6.3.3 Época

A época do tempo base é a origem do tempo como medida do sistema, que é o tempo zero. Como na exactidão, a época do tempo base é determinada pelo GrandMaster. Em muitas aplicações de teste e medida só o tempo relativo dentro do sistema é importante e não é necessária a correspondência ao tempo civil. Outros sistemas requerem um tempo base para serem detectáveis para UTC (*Coordinated Universal Time*). Este é actualmente, relativamente fácil de implementar e possui vantagens como tornar os dados de engenharia correlacionáveis com dados de outros sistemas ou com dados históricos.

A maneira mais simples de igualar o tempo base do IEEE 1588 com o do UTC é sincronizar o relógio GrandMaster a uma fonte UTC detectável, tal como um sistema GPS ou um servidor de tempo NTP. Isto deve ser feito com o cuidado de reter a precisão desejada para o tempo base resultante. A precisão NTP é na ordem dos milisegundos, portanto só com muitas médias e depois de longos períodos tempo é que estes valores devem ser usadas para sincronizar o GrandMaster no IEEE 1588. A precisão de GPS é muito melhor, na ordem dos 50 ns, mas ainda requer médias para uma melhor exactidão. Ambos, NTP e GPS, permitem o salto necessário para traduzir o tempo UTC em tempo base do IEEE 1588.

É interessante referir que os grandes fornecedores em indústrias de automação industrial e potência com o IEEE 1588 implementado, têm relatado o sucesso da ligação do tempo base do IEEE 1588 para UTC por meio de GPS, dado que muitos dos seus consumidores

têm frequentemente exigências para ter os *timestamps* a eventos nos seus sistemas em UTC [Eid05c].

2.6.3.4 Dependência da topologia da rede

O protocolo IEEE 1588 é um protocolo de sincronização *Master-Slave*. Cada *Slave* ajusta o seu tempo base para estar de acordo com o seu *Master*. A exactidão relativa entre o *Master* e o *Slave* será variável com o tempo devido à natureza do *servo loop* no *Slave*, as flutuações nos osciladores dos dois relógios, e o mais crítico as flutuações introduzidas pelos dispositivos da rede. Sistemas em que as comunicações entre dispositivos atravessam *switches* ou *routers*, como é o caso da Ethernet, introduzem flutuações que não sendo corrigidas irão limitar a precisão e exactidão dependendo dos padrões de tráfego na rede. A solução é modificar estes dispositivos para incluírem relógios IEEE 1588 servindo como transferência do modelo. Estes dispositivos presentes teriam de ser especiais tal como vêm especificados na norma IEEE 1588, sendo chamados de *boundary clocks*.

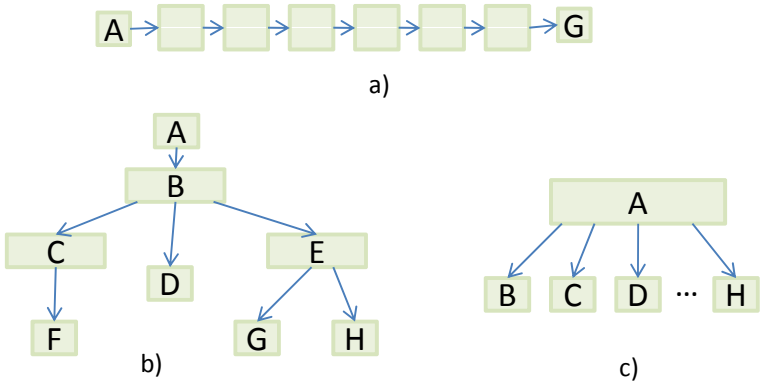


Figura 2.11: Topologias de rede com o IEEE 1588

Os sistemas IEEE 1588 complexos terão hierarquias que resultam em *servo loops* em cascata, levando à degradação do tempo base. Por exemplo, na Figura 2.11 a) a imprecisão entre os relógios A e G na topologia linear irá ser maior do que na topologia de ramos na Figura 2.11 b), já que no caso da linear existem sete *control loops* em cascata envolvidos enquanto na topologia da árvore só três desses *loops*.

Também é claro que a colocação de relógios com diferentes precisões é importante. Por exemplo, se na topologia linear da Figura 2.11 algum dos relógios, entre os relógios A e G, tiver uma precisão significativamente baixa em relação aos outros, a precisão relativa de A para G será no máximo a mesma do que a do relógio de baixa precisão. Na topologia de árvore é possível isolar os relógios de baixa precisão num dos ramos da árvore preservando desse modo a precisão dos restantes ramos da árvore.

É imperativo que os requisitos de precisão de cada relógio, do *switch* IEEE 1588 e todos os sistemas de precisão sejam cuidadosamente considerados aquando do desenho da topologia da rede. Para os sistemas mais complexos a topologia será parecida com a topologia em árvore. Para sistemas muito exigentes do ponto de vista do rigor, é muito provável que na topologia apareça um único *switch* IEEE 1588 activo e com poucos dispositivos conectados a ele como mostra na Figura 2.11c).

Os sistemas em aplicações de industriais de automação são tipicamente bastante complexos e envolvem 1000 ou mais sensores ou actuadores distribuídos e múltiplos níveis de controlo. Neste caso são usadas as topologias em árvore e a linear. O uso da topologia linear levou ao desenvolvimento de uma nova tecnologia que incorpore o IEEE 1588 nos vários dispositivos da rede. Estes dispositivos designados *transparent clocks*, foram demonstrados por um fornecedor numa conferência do IEEE 1588 em 2004 [VS04], sendo que as especificações deste dispositivo são susceptíveis de serem adicionadas na próxima revisão da norma.

2.6.3.5 Recursos necessários

O IEEE 1588 é um protocolo muito leve. No caso normal só um ou dois pacotes por segundo passam no *link* entre um relógio *Master* e um *Slave*. A memória e os recursos computacionais necessários também são baixos. Contudo é importante para ambos os dispositivos e para os desenhadores do sistema, garantir que os recursos fornecidos são suficientes para responder às exigências totais da aplicação sem que o protocolo sinta falta de recursos.

A principal preocupação é a transferência de grandes quantidades de dados entre os dispositivos. Embora com o devido projectar de relógios a perda ocasional de mensagens IEEE 1588 não seja muito significativa, é importante que o sistema não cause múltiplas e sucessivas perdas. Se a rede ou as capacidades do sistema não forem adequadas e se o tráfico IEEE 1588 não tiver prioridade suficiente, poderá ocorrer a degradação da precisão do tempo.

2.6.3.6 Comportamento dinâmico

Em sistemas IEEE 1588 cada relógio *Slave* sincroniza para o seu *Master* tipicamente com um simples algoritmo de controlo tal como um *loop* proporcional integral (PI). O objectivo do algoritmo de controlo é trazer o relógio para o estado sincronizado e eliminar o ruído que de outra forma provocaria a perturbação do tempo base. Em geral, para situações em que o ruído seja maior torna-se necessário efectuar mais medições para só depois actuar no sistema. Para conseguir obter boas precisões são necessários alguns componentes que tenham osciladores mais estáveis e *Slaves* que usem médias de tempos longos. Como resultado os tempos de inicialização ou transiente serão mais longos. Para atenuar este problema é possível usar controladores mais sofisticados (ex. que ajustem as constantes de tempo dinamicamente).

2.7 Conclusão

Os protocolos de sincronização referidos neste capítulo apresentam-se como os mais utilizados em redes de comunicação. Contudo mediante os requisitos temporais do sistema, deve-se escolher em conformidade o melhor método para sincronizar os relógios. É relativamente fácil perceber que em sistemas com baixos requisitos temporais, o NTP ou o IEEE 1588 baseado apenas em software são boas opções. Já para sistemas de tempo-real com exigências temporais apertadas em que os vários dispositivos necessitem de ter relógios rigorosos, terão de ser usados o SynUTC ou o IEEE 1588 assistido por hardware.

Actualmente o NTP é largamente utilizado na Internet, estando por isso bem estabelecido. Na Internet o NTP é utilizado em aplicações que não necessitem de grandes restrições temporais, conseguindo-se obter precisões de dezenas de milisegundos. Segundo a norma, o *offset* entre o cliente NTP e o servidor nunca deve ser superior a 128 ms, pois nesses casos o processo de sincronização pode-se tornar muito longo ou nunca acontecer.

Outro senão, reside no facto de o NTP apenas ser desenhado para poder ser aplicado em Ethernet, enquanto o IEEE 1588 e o SynUTC prevêem a possibilidade de implementação em outras redes.

Já o IEEE 1588 foi originalmente introduzido nas industrias de teste e automação, conseguindo precisões na ordem dos sub microsegundos. Mais tarde ganhou popularidade e outros grupos mostraram interesse, como as telecomunicações, a distribuição de potência eléctrica e as aplicações militares. Este também tem grande utilidade em aplicações na Internet que necessitem multimédia de tempo-real rigoroso ou em aplicações de controlo.

Com IEEE 1588 é possível conseguir precisões na ordem dos nanosegundos, obviamente com soluções assistidas por hardware. O SynUTC é um protocolo relativamente recente que permite sincronizações da mesma ordem que o IEEE 1588, apresentando algumas vantagens e desvantagens relativas ao IEEE 1588.

O SynUTC apresenta vantagens ao nível da tolerância a falhas (dado que não utiliza um principio *Master-Slave*), e suporta sincronizações de relógios externos. O IEEE 1588 feito para redes pequenas (não a nível global), consome poucos recursos, utiliza o principio de *Master-Slave*. Este possui mecanismo de particionamento automático da rede e de selecção do melhor relógio na rede. Permite implementações feitas somente em software (sem alteração de hardware), algo que não é previsto para o SynUTC. No entanto o mecanismo de tolerância a falhas previsto norma do IEEE 1588 não é tão robusto quanto o do SynUTC. Contudo na conferência do IEEE 1588 em 2006 foi proposto um mecanismo de tolerância para soluções em hardware de grande precisão [LGK06].

O IEEE 1588 apresenta uma vantagem não menos relevante, que é o facto de já ter conquistado o mercado, sendo que actualmente existem inúmeros equipamentos que assentam o seu método de sincronização, enquanto que o SynUTC teve uma origem académica, sendo que só agora começa a oferecer as primeiras soluções comerciais.

Capítulo 3

Sincronização de relógio em CAN

3.1 Indrodução

Neste capítulo apresenta-se alguns trabalhos encontrados sobre sincronização de relógio em CAN. Também será referida a adaptação do protocolo IEEE 1588 para *DeviceNet*, dado que este protocolo de comunicação tem o CAN como sua espinha dorsal. No entanto, primeiramente será feita uma abordagem ao protocolo CAN e as suas principais características.

3.1.1 *Controller Area Network*

O protocolo *Controller Area Network* (CAN) desenvolvido pela Robert Bosh GmbH, oferece uma solução robusta para a comunicação entre dispositivos. A comunicação pode ocorrer num máximo absoluto até 1 Mbit/s (num barramento com cerca de 40 m de comprimento). Apesar de ter sido desenvolvido para aplicações automóveis, é também actualmente usado em aplicações de automação industrial devido ao seu baixo custo, alta performance, baixos tempos de latência e flexibilidade de configuração.

O CAN é um CSMA/CA (*Carrier Sense Multiple Access / Collision Avoidance*), possui uma arquitectura *multi-Master*, mensagens até 8 bytes de dados com prioridades que são enviadas pelo barramento série. O protocolo CAN possui um mecanismo cómodo em que com 15 bits do campo CRC garantem uma elevada integridade dos dados transmitidos.

O protocolo CAN prevê quatro benefícios principais. Primeiro, o protocolo simplifica e economiza as tarefas de interface dos subsistemas de diferentes fornecedores numa rede comum. Em segundo lugar, a carga das comunicações é transferida do CPU anfitrião para os periféricos inteligentes, tendo assim o CPU anfitrião mais tempo para correr as tarefas do sistema. Em terceiro lugar, como rede multiplexada, o CAN reduz enormemente o tamanho de cabo utilizado e elimina as ligações ponto-a-ponto. Por último, como um protocolo *standard*, o CAN tem amplo mercado que motiva os fabricantes destes semicondutores a desenvolver dispositivos CAN a preços competitivos.

. Arbitragem não destrutiva no acesso ao meio

O protocolo CAN implementa o acesso prioritizado ao meio. Cada mensagem tem um único identificador, que determina a sua prioridade em relação às outras. Quando o barramento está livre, qualquer controlador CAN está autorizado a transmitir. Se mais do que um controlador começar a transmitir em simultâneo, segue-se uma comparação bit a bit aos identificadores das mensagens. No final só a mensagem mais prioritária

será transmitida, enquanto os restantes nodos perderão a arbitragem e não tentarão a transmissão novamente da mensagem até que o barramento esteja livre.

. **Visão simultânea dos bits**

Para permitir a comparação bit a bit dos identificadores das mensagens, os controladores CAN devem ter a mesma visão do bit que está no barramento. Isto implica, primeiro, existir uma relação específica entre a taxa de transmissão e o comprimento do barramento [eA01], e segundo, ter um mecanismo de sincronização de bit que garante que todos os controladores CAN fazem a amostragem do bit num instante quase em simultâneo [fhsc93]. A localização do ponto de amostragem é mostrado na Figura 3.1.

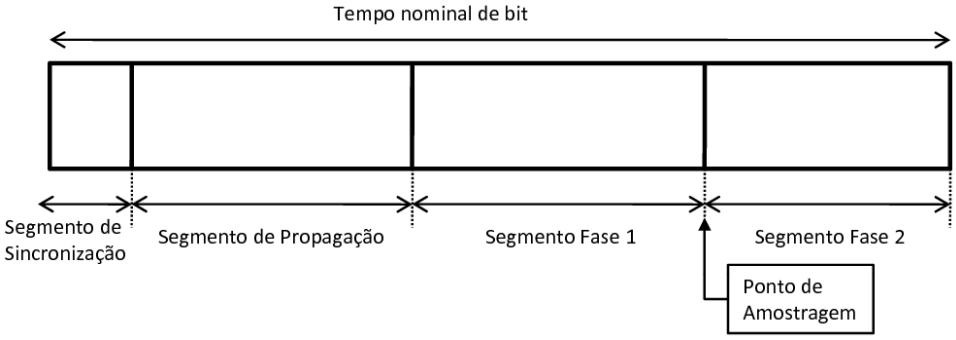


Figura 3.1: Local onde é feita a amostragem no tempo de bit CAN.

O segmento de sincronização e de propagação tem um comprimento constante, enquanto o segmento fase 1 e o segmento fase2 podem ser alargados ou estreitados, respectivamente, seguindo as seguintes regras: o controlador CAN receptor deve alargar a fase 1 se o seu oscilador local for mais rápido que o oscilador local do controlador CAN transmissor; em contraste, o controlador receptor CAN deve estreitar o segmento fase 2 se o seu relógio local for mais lento que o oscilador local do controlador CAN transmissor. Actuando desta forma, o desvio entre os relógios locais será compensado e por isso os pontos de amostragem dos vários controladores CAN serão muito próximos. Como referido em [RGR98], esta diferença é da ordem do atraso de propagação que representa 10-30% do tempo de bit.

. ***Broadcast* atómico**

As capacidades do protocolo CAN na detecção de erros e sinalização dos mesmos, permite que qualquer trama que é enviado através da rede CAN, é em princípio consistentemente recebido por todos os controladores da rede ou por nenhum deles, apesar de estarem descritos na literatura cenários de inconsistência [RVA+98]. Além disso, por causa da retransmissão automática de tramas que o CAN implementa, qualquer mensagem emitida na rede CAN é correctamente recebida por pelo menos um nodo, enquanto o controlador transmissor permanecer num estado sem falhas.

3.1.2 Propriedades do CAN na sincronização de relógio

Num sistema distribuído a sincronização de relógio depende da troca de mensagens no barramento bem como do rigor temporal com que os eventos de transmissão e recepção são

medidos. Assim, as condições de carga na rede e nos CPU's limitam a precisão do algoritmo de sincronização de relógio.

Por exemplo, num barramento CAN se existir uma grande carga de mensagens com prioridade superior às mensagens de sincronização, o tempo entre sincronizações válidas poderá nestas situações ser maior que o desejado.

Assim, para garantir que o intervalo entre sincronizações é respeitado será necessário uma análise temporal adequada [TBW95]. Ao contrário de outras redes, em CAN é possível determinar analiticamente o tempo de resposta de uma mensagem mediante as condições do barramento, tornando-se assim relativamente mais fácil analisar o comportamento das mensagens do protocolo de sincronização mediante diferentes condições de carga.

Muitas das aplicações CAN são baseadas em sistemas distribuídos embutidos de baixo custo, que ao contrário de outros tipos de sistemas têm fortes limitações na capacidade de processamento ao nível dos nodos, bem como na largura de banda disponível na rede.

Estas limitações fazem com que os algoritmos a aplicar a este tipo de sistemas não devam exijir demasiada capacidade de processamento e que a troca de mensagens necessária para a sincronização dos vários dispositivos da rede não gaste demasiada largura de banda.

Assim neste capítulo irão ser brevemente explicadas as principais soluções já encontradas no intuito de resolver este problema. Primeiramente descreve-se a adaptação do IEEE 1588 em *DeviceNet*, dado que o *DeviceNet* é um protocolo de comunicação que assenta sobre a rede CAN. Posteriormente, vai-se falar dos vários protocolos encontrados como estado da arte sobre sincronização de relógio em CAN, conseguidos através de diversas técnicas e algoritmos.

3.2 Adaptação do IEEE 1588 para *DeviceNet*

O protocolo IEEE 1588 foi pensado para ser independente da rede em que é aplicado. Cada rede que suporte o IEEE 1588 prevê uma definição dentro de um anexo da norma sobre a forma de mapear o 1588 na rede. Na norma actual apenas existe a definição da rede para a Ethernet (UDP/IP).

Contudo, a ODVA (*Open DeviceNet Vendor Association*) tem feito grande esforço para que a corrente norma tenha um anexo para *DeviceNet*. A ODVA é a responsável pela afirmação e controlo da especificação *DeviceNet*, tendo 350 companhias membros à volta do mundo, estimando-se que estejam instalados 8 milhões de nodos *DeviceNet* [ODV03].

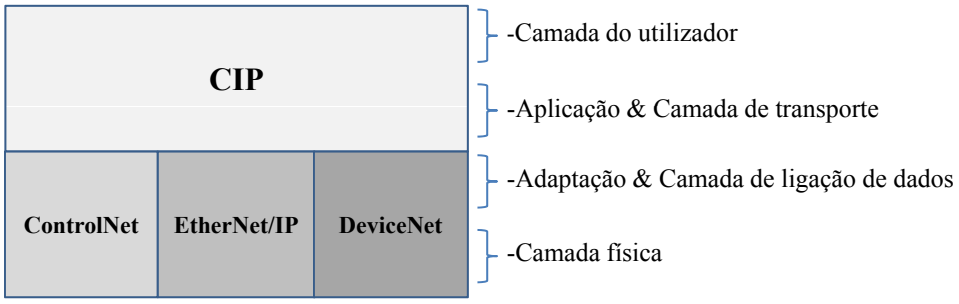


Figura 3.2: Arquitectura CIP

Neste seguimento, o CIP (*Common Industrial Protocol*) pretende introduzir nas suas redes (*DeviceNet, ControlNet e EtherNet/IP*) uma aplicação de cobertura designada por *CIP*

sync que permita uma sincronização melhor que 500 nanosegundos; tolerância a sistemas heterogéneos com relógios de várias precisões; resoluções e estabilidade e que use o mínimo de largura de banda, computação e recursos da memória do nodo.

O *DeviceNet* é uma rede digital, *multi-drop* que conecta e serve como rede de comunicação entre controladores industriais e dispositivos I/O. Cada dispositivo e/ou controlador é um nodo da rede. Este protocolo usa o CAN como espinha dorsal, dado que o *Data Link Layer* do *DeviceNet* é definido pela especificação CAN e implementado por controladores CAN.

O *DeviceNet* é uma rede produtor-consumidor que suporta múltiplas comunicações hierárquicas e prioridades de mensagens. Os sistemas *DeviceNet* podem ser configurados para operar numa arquitectura *Master-Slave* ou controlo distribuído usando comunicação *peer-to-peer*.

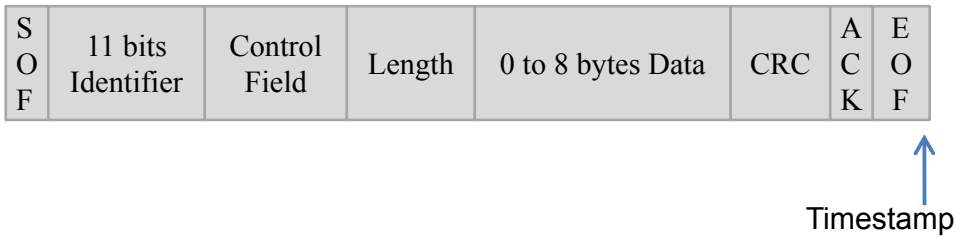


Figura 3.3: *Timestamp* da trama *DeviceNet*.

O protocolo *DeviceNet* permite até 64 dispositivos (nodos) numa sub-rede. A cada dispositivo é atribuído parte do espaço de identificação de mensagens (2032 mensagens). É necessária a fragmentação para conseguir o envio de mensagens grandes usando para isso múltiplas tramas CAN.

A adaptação do 1588 para *DeviceNet* consiste em quatro aspectos essenciais:

. **Selecção do ponto de *timestamp* da mensagem**

  O ponto de *timestamp* da mensagem corresponde à borda do sexto bit do campo EOF (*End Of Frame*), do primeiro pacote fragmentado da mensagem PTP.

. **Definição do UUID**

  O *DeviceNet* (assim como todas as redes dentro do CIP) partilha um único ID de vendedor com 16 bits para cada construtor dos produtos CIP. Em adição todos os vendedores atribuem um número de série de 32 bits para cada produto CIP fabricado. Assim a combinação do ID do vendedor CIP e número de série é universalmente única.

  Dado isto, o UUID (identificador universalmente único) no 1588 para a adaptação em *DeviceNet* deve ser o ID do vendedor, o número de série do produto e ainda o ID do nodo *DeviceNet* em questão.

. **Mapeamento dos portos 1588 e dos endereços *multicast***

  Os endereços *multicast* do PTP devem ser mapeados em identificadores de mensagem *DeviceNet*, permitindo assim um código único pela combinação do endereço do subdomínio PTP e o porto PTP.

  Para poupar largura de banda, todas as mensagens PTP que não sejam *Management* devem usar o valor do subdomínio codificado em 8 bits dentro do cabeçalho PTP para representar o nome do subdomínio PTP.

UUID_field = Vendor Id +
Product Serial Number +
Node Id

Figura 3.4: Parâmetros que constituem o identificador único universal em *DeviceNet*.

. Definição do formato das mensagens

Todas as mensagens PTP que não sejam *Managment* são empacotadas, ao contrário da Ethernet que tem 32 bits de alinhamento. Assim, por exemplo, a mensagem *Sync* será enviada em 15 tramas CAN e a mensagem *Follow-up* em 5 tramas CAN.
As mensagens PTP *Managment* são consistentes com as mensagens usadas na Ethernet para reduzir a carga e encaminhar os problemas de compatibilidade para os *boundary clocks* (*routers*).

3.3 Protocolos de sincronização de relógio em CAN

Seguidamente serão brevemente descritos os principais protocolos de sincronização de relógio em CAN encontrados na literatura científica:

3.3.1 S1. *Implementing a Distributed High Resolution Real-Time Clock using the CAN-bus (1994)*[GS94]

É um protocolo simples que sincroniza os relógios locais dos nodos ligados ao barramento de campo, especialmente ao barramento CAN. Todos os nodos participantes têm relógios locais com possibilidade de terem diferentes exactidões. Este relógio local é usualmente baseado no *timer* dos microcontroladores, isto em sistemas embutidos de baixo custo. O algoritmo assenta num esquema *Master/Slave* não apresentando nenhum mecanismo de tolerância a falhas, tendo por isso algumas restrições para o seu correcto funcionamento.

O grupo de tempo-real com o projecto *GMDs CREW* implementou este protocolo de sincronização de relógio no barramento CAN que fornecia um tempo global comum com uma precisão de 20 microsegundos. O protocolo é simples e totalmente independente do hardware e utiliza pouca largura de banda (menos de 20 mensagens/segundo).

Esta implementação utilizou microcontroladores (Intel 8051) ligados num barramento CAN a 1Mbit/s. Os microcontroladores corriam com cristais de 16MHz e outros com 12 MHz, resultando numa resolução máxima de relógio de 0.75 e 1 microsegundo, respectivamente. Finalmente o nodo *Master* estava ligado a um receptor GPS, tendo por isso o tempo global.

3.3.2 S2. *Fault-tolerant Clock Synchronization in CAN (1998)*[RGR98]

Este algoritmo foi inspirado num algoritmo genérico para redes *broadcast* [VR92, VRC97], sendo que este ultrapassa muitas das limitações impostas por esse algoritmo genérico ao nível da tolerância a falhas, na exactidão da sincronização de relógio conseguida e no número de mensagens que necessita de trocar no processo de sincronização.

Este algoritmo designado por *phase-decoupled* oferece uma precisão estreita e uma boa exactidão com um custo razoável. Por exemplo, para se obter uma precisão de 100 μs, os relógios têm de ser sincronizados apenas uma vez a cada 45s e a perda de exactidão é apenas na ordem de 4.2ms por hora.

Importa ainda referir que este algoritmo permite a sua implementação apenas em software para a precisão anteriormente referida, assenta num esquema simétrico, tolerando qualquer falha de relógio e de mensagens inconsistentes.

3.3.3 S3. e S4. *Time Triggered Communication on CAN (2000)*[HMF+00]

O TTCAN é um protocolo que se encontra standardizado pela *International Standardisation Organisation* (ISO) com a referência ISO 11898-4 [HMF+00]. Este surgiu sobretudo pelo facto de a comunicação numa rede CAN clássica ser *event-triggered*, podendo assim ocorrer picos de carga no barramento quando várias mensagens são pedidas para enviar ao mesmo tempo. Mesmo as mensagens não sendo perdidas, dado que o CAN tem uma arbitragem não destrutiva que garante o envio sequencial de todas as mensagens mediante o seu identificador de prioridade, num sistema de tempo-real em que as transmissões têm *deadlines* a cumprir, o envio das mensagens de menor prioridade certamente não será respeitado em situações críticas.

Neste contexto, foi desenvolvido o *Time-Triggered* CAN implementado em 2 níveis. O nível 1 é limitado apenas à transferência cíclica de mensagens, garantindo um padrão de comunicação no barramento determinístico, conseguindo assim um melhor aproveitamento da largura de banda da rede CAN. O nível 2 além do que é suportado no nível 1, também suporta um sistema de tempo global, permitindo uma visão do tempo sincronizado em todo o sistema e uma contínua correcção do *drift* nos controladores CAN.

O nível 1 pode ser implementado apenas em software. Mas quando as taxas de transmissão são altas e existe um grande número de mensagens a enviar, a carga no CPU será alta. Assim nestes casos deve-se escolher uma solução em hardware.

O *Time-Triggered* CAN assenta num esquema *Master/Slave* tolerando a falha do nodo *Master*, mas não tolera falhas de performance dos relógios, nem a omissão de mensagens inconsistentes.

Assim no TTCAN nível 1 consegue-se uma precisão da ordem dos milisegundos numa implementação baseada em software, enquanto no TTCAN nível 2 consegue-se uma precisão na ordem dos 10 microsegundos numa solução implementada em hardware.

3.3.4 S5. Hardware Design of a High-precision and Fault-tolerant Clock Subsystem for CAN Networks (2003)[RNBP03]

Esta arquitectura assenta num subsistema de relógio em hardware que permite às redes CAN terem um serviço de sincronização de relógio. Esta implementação apresenta significativas vantagens em relação às soluções anteriormente implementadas:

i) Ortogonalidade, visto que não necessita de nenhum componente da rede especifico, sendo compatível tanto com comunicação *event-triggered* como *time-triggered*.

ii) Consegue-se obter uma boa precisão na sincronização de relógio tirando partido das boas propriedades do protocolo CAN. O algoritmo utilizado é baseado no TTCAN nível 2 [HMF+00], mas com algumas mudanças na correcção do *drift*.

iii) As unidades de relógio apresentam um comportamento de tolerância a falhas. Esta propriedade é conseguida através da duplicação da unidade de relógio e de um algoritmo distribuído, que permitem saber a actual disponibilidade de todas as unidades de relógio relevantes.

Esta arquitectura efectua o *timestamp* via hardware, utiliza um conceito *Master/Slave*, tolera falha dos relógios e certas falhas de performance. A 1Mbit/s este desenho permite obter precisões de 10 microsegundos quando sincronizado todos os segundos.

3.3.5 S6. *Fault-tolerant Clock Synchronisation with microsecond-precision for CAN Networked Systems (2003)* [LA03]

Esta arquitectura foi desenvolvida por *DRTS Ltd*, consistindo numa técnica de sincronização de relógio tolerante a falhas e baseada em software para redes *broadcast*, tal como o CAN. O protocolo fornece um serviço previsível, confiável, robusto e permite a sincronização de sistemas de rede com precisão de microsegundo usando uma largura de banda desprezável.

Esta abordagem é fácil de implementar não necessitando de adicionar novo hardware. Permite que as comunicações *time-triggered* sejam implementadas em nodos CAN *standard*.

Este algoritmo permite obter precisões dos 10 microsegundos, assenta numa estrutura *Master/Slave* que o torna fácil de implementar. Este apresenta-se como uma solução de baixo custo para aplicações de segurança crítica, tendo uma boa eficiência de carga na rede (0.1% a 1Mbit/s) e apresentando grande flexibilidade, sendo que é possível em qualquer instante adicionar ou remover nodos *Slave* ou *Master* com a rede em funcionamento.

3.4 Conclusão

Relativamente à adaptação do IEEE 1588 para *DeviceNet*, verifica-se que a implementação deste protocolo na transmissão de diversas tramas CAN que são necessários enviar por cada mensagem do PTP, o que além de representar uma significativa ocupação do barramento (em redes com baixas taxas de transmissão), representam uma adição substancial de carga no CPU para o atendimento dessas mensagens.

Assim, pode considerar-se que esta implementação implica um *overhead* elevado quer em termos de largura de banda que em termos de processamento, factores especialmente relevantes em sistemas distribuídos de baixo custo.

Para além do *DeviceNet*, encontraram-se outros protocolo de sincronização de relógio para redes CAN, verificando-se a existência de diferenças ao nível das técnicas aplicadas, bem como nos requisitos que cumprem. Este facto, juntamente com a complexidade inerente dos problemas da sincronização de relógio, fazem com que a comparação entre implementações seja uma tarefa relativamente difícil.

Para facilitar essa comparação em [RNP03] é sugerido uma representação tridimensional dos sistemas, em que cada eixo representa as propriedades mais relevantes para o serviço de sincronização de relógio em sistemas distribuídos embutidos de baixo custo, nomeadamente o rigor, fiabilidade e eficiência de custos. Este método de comparação apresenta-se como uma boa forma de equiparar soluções e demonstra de uma maneira intuitiva formas de melhorar as soluções. A Figura 3.5 ilustra a representação das várias soluções descritas anteriormente para a sincronização de relógio em CAN utilizando este sistema de representação.

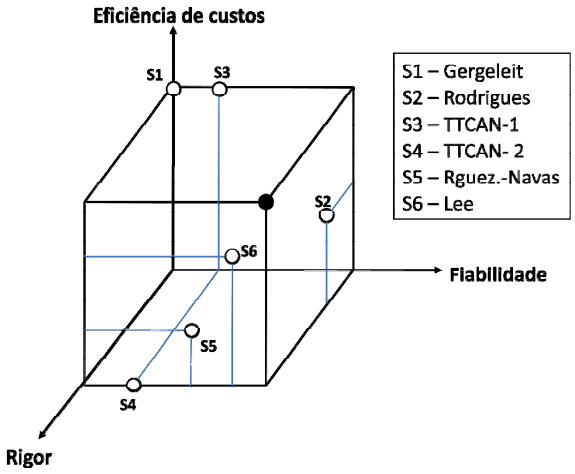


Figura 3.5: Representação das soluções para sincronização de relógio em CAN.

Analisando a figura é possível concluir que as técnicas já propostas, deixam espaço aberto para se encontrarem novas soluções que de alguma forma tentem cumprir melhor as propriedades relevantes neste tipo de sistemas. De facto algumas das técnicas empregues são rigorosas mas significativamente dispendiosas, outras são fiáveis mas têm pouco rigor, percebendo-se assim que nenhuma das implementações representadas no cubo cumpre as três propriedades em simultâneo (representado pelo ponto a cheio situado num dos vértices).

A implementação do protocolo de sincronização realizada no âmbito desta dissertação enquadra-se neste cubo como tendo grande rigor, sendo pouco dispendiosa e apresentando uma fiabilidade relativamente baixa, no caso de não serem utilizados mecanismos de tolerância a falhas.

Capítulo 4

Implementacão em Software

4.1 Introdução

Neste capítulo serão indicadas as adaptações efectuadas ao protocolo IEEE 1588 (abordado na secção 2.6), pois este tal como vem definido na norma torna-se demasiado pesado num ambiente de microcontroladores de baixa performance e com pouca largura de banda. Seguidamente será abordada a implementação em software do protocolo de sincronização em CAN. A plataforma de desenvolvimento utilizada é baseada nas placas DETPIC, integrando um PIC18F258, com controlador CAN incorporado. A partir das bibliotecas disponibilizadas, em linguagem "C"para acesso ao controlador CAN, será implementado o protocolo ao nível da interrupção, sendo utilizado o compilador HI-TIDE versão estudante da Hi-Tech.

4.2 Especificações do protocolo implementado - 1588Light

O protocolo implementado é baseado no protocolo IEEE 1588 [IEE02]. Mas devido ao elevado número de campos necessários enviar em cada mensagem do protocolo, fez-se uma adaptação deste para a rede CAN, passando a designar-se por 1588Light. No anexo A.4 é possível ver os campos que compõem cada mensagem do PTP segundo a norma. Estes iriam preencher um grande número de tramas CAN, tal como já acontece também na adaptação do IEEE 1588 ao *DeviceNet* (ver em 3.2). Mas dado que os sistemas CAN a aplicar este novo protocolo são sistemas distribuídos de baixo custo e com baixa capacidade de processamento, optou-se por tornar o protocolo mais leve quer ao nível de carga no barramento, quer ao nível de carga de processamento.

4.2.1 Mensagens

Cada mensagem do 1588Light irá ocupar apenas uma trama CAN, sendo cada mensagem caracterizada tal como vem apresentado na Tabela 4.1. Neste protocolo apenas é enviado o conteúdo estritamente necessário e relevante para a sincronização do relógio, ou seja, os *timestamps* efectuados do lado do relógio *Master*. Foram eliminados todos os campos da norma 1588 respeitantes ao algoritmo de selecção automática do relógio *Master* na rede, bem como aos que se revelem pouco importantes para sincronizações relógio em CAN. Assim sendo, parte-se do princípio que a rede já tem um *Master* atribuído por defeito, não sendo por isso necessário um algoritmo de selecção do *Master*.

Mensagem	Emissor	Bytes de Dados	RTR	Conteúdo
Req_Sync	*Slave*	0	1	-
Sync	*Master*	0	0	-
Follow_up	*Master*	8	0	*Timestamp* do envio da mensagem *Sync*
Delay_Req	*Slave*	0	0	-
Delay_Resp	*Master*	8	0	Timestamp de recepção da mensagem *Delay_Req*

Tabela 4.1: Campos das mensagens do 1588Light.

Dado a simetria do barramento CAN, não é necessário repetir as mensagens *Sync* e *Follow_up* no processo de sincronização, tal como é previsto acontecer na em redes assimétricas.

Este novo protocolo também apresenta uma particularidade diferente do IEEE 1588. Enquanto que no IEEE 1588 é o relógio *Master* que envia periodicamente em *broadcast* as mensagens *Sync* e *Follow_up* para sincronizar todos relógios *Slave* existentes na rede, no 1588Light são os relógios *Slaves* que decidem quando querem sincronizar. Os *Slaves* são responsáveis por desencadear o processo de sincronização com o *Master* através do envio da mensagem *Req_Sync*, permitindo uma maior flexibilidade e evitando grande picos de processamento no relógio *Master*, tal como vai ser explicado seguidamente.

4.2.2 Carga no barramento

Ocupação do barramento quando é o *Master* a despoletar o processo de sincronização (U_{Sync}^{MST}):

$$U_{Sync}^{MST}(n) = \frac{t\_Sync + t\_Follow\_up + n \times (t\_Delay\_Req + t\_Delay\_Resp)}{\Delta T} \times 100 \qquad (4.1)$$

Ocupação do barramento quando é o *Slave* a despoletar o processo de sincronização (U_{Sync}^{SLV}):

$$U_{Sync}^{SLV}(n) = \frac{n \times (t\_Req\_Sync + t\_Sync + t\_Follow\_up + t\_Delay\_Req + t\_Delay\_Resp)}{\Delta T} \times 100 \qquad (4.2)$$

Em que:

1. *t_Sync* é o tempo de transmissão de uma mensagem *Sync*.
2. *t_Follow_up* é o tempo de transmissão de uma mensagem *Follow_up*.
3. *t_Delay_Req* é o tempo de transmissão de uma mensagem *Delay_Req*.
4. *t_Delay_Resp* é o tempo de transmissão de uma mensagem *Delay_Resp*.
5. ΔT é o intervalo de tempo entre sincronizações.

	Ocupação do barramento	
Nº *Slaves*	***Master* a iniciar**	***Slave* a iniciar**
1	0.019%	0.022%
4	0.048%	0.087%
8	0.086%	0.174%
16	0.162%	0.348%

Tabela 4.2: Ocupação do barramento imposta pelo protocolo de sincronização a 1Mbit/s com intervalo de sincronização igual a 2 segundos.

Apesar de este método indicar uma carga de barramento potencialmente superior, o facto de ser o *Slave* a desencadear o processo de sincronização, permite que cada um efectue o processo de sincronização conforme as suas necessidades e restrições temporais. Assim por exemplo, um relógio *Slave* que não necessite de um tempo muito rigoroso poderá sincronizar-se apenas de quatro em quatro segundos, enquanto que um relógio *Slave* que tenha necessidade de ter um tempo bastante exacto pode sincronizar-se todos os segundos. Neste mesmo exemplo, se o método utilizado fosse o relógio *Master* a desencadear o processo de sincronização, seria necessário que o *Master* sincronizasse periodicamente à frequência de pior caso, ou seja, sincronizasse todos os segundos. Assim contrariando o que já anteriormente foi referido, este método acabaria por gastar uma maior percentagem de barramento em relação ao método em que é o *Slave* a despoletar o processo de sincronização ($0.057\% > 0.054\%$).

Além disso, o facto de os vários *Slaves* se sincronizarem mediante as suas necessidades atenua estatisticamente a existência de grandes picos de carga de processamento no *Master*. No IEEE 1588 esses resultam do facto de todos os *Slaves* receberem as mensagens *Sync* e *Follow_up* no mesmo instante, fazendo com que todos tentem de seguida enviar para o relógio *Master* a mensagem *Delay_Req*. A recepção destas mensagens no *Master* levará a um pico de carga no CPU, devido ao atendimento e processamento das mesmas.

Por fim, tal como já foi indicado, sendo o *Slave* a desencadear o processo de sincronização consegue-se uma maior flexibilidade. Esta flexibilidade resulta de podermos adicionar ou remover nodos (com diferentes frequências exigidas para realizar o processo de sincronização) sem termos de alterar o sistema. Por exemplo, se numa rede existirem 2 nodos *Slave* com necessidade de se sincronizarem de 4 em 4 segundos, ao ser adicionado um nodo *Slave* com restrições temporais mais apertadas e com necessidade de sincronizar todos os segundos, pelo método adoptado não seria necessário efectuar qualquer alteração no sistema. Já se o *Master* fosse responsável por desencadear o processo de sincronização, levaria à necessidade de este ser ajustado na frequência com que desencadeia o processo sincronização sempre que fossem adicionados ou removidos nodos *Slaves* com diferentes necessidades para a realização do processo de sincronização.

4.2.3 Gama de identificadores

As mensagens do 1588Light são caracterizadas por uma gama de IDs específica, permitindo assim distingui-las das outras mensagens que circulam no barramento. Além do objectivo de distinção, uma gama de IDs específica permite dar uma importância às mensagens do protocolo no barramento, podendo ter impacto quando o barramento se encontra bastante ocupado. A gama reservada para identificadores do protocolo é composta por um valor fixo

nos cinco bits mais significativos do ID e um valor variável para os restantes seis bits. Os bits variáveis permitirão identificar cada um dos relógios *Slave* na rede, sendo por isso possível definir uma rede com

$$n^{o}\ bits\ variaveis = 11 - 5 = 6\ bits \qquad 2^{6} = 64\ nodos\ Slave \tag{4.3}$$

Assim sendo, e utilizando apenas 64 IDs da gama total de identificadores, consegue-se um número apreciável de *Slaves* a sincronizar num único relógio *Master*. No entanto se pretendermos mais relógios *Slave* a sincronizar numa mesma rede, basta apenas alargar a gama de IDs reservados. Por exemplo, se o valor fixo for os 3 bits mais significativos e os restantes oito bits forem variáveis torna-se possível definir numa rede

$$n^{o}\ bits\ variaveis = 11 - 3 = 8\ bits \qquad 2^{8} = 256\ nodos\ Slave. \tag{4.4}$$

4.2.3.1 Identificador *Master/Slave*

Neste protocolo o ID produzido pelo *Slave* será o mesmo que o *Master* utilizará para responder a esse *Slave*. Este facto não levanta problemas dado que é garantido que o *Master* e o *Slave* não irão concorrer no envio de mensagens com o mesmo identificador. Isso mesmo pode ser visto na Figura 4.1.

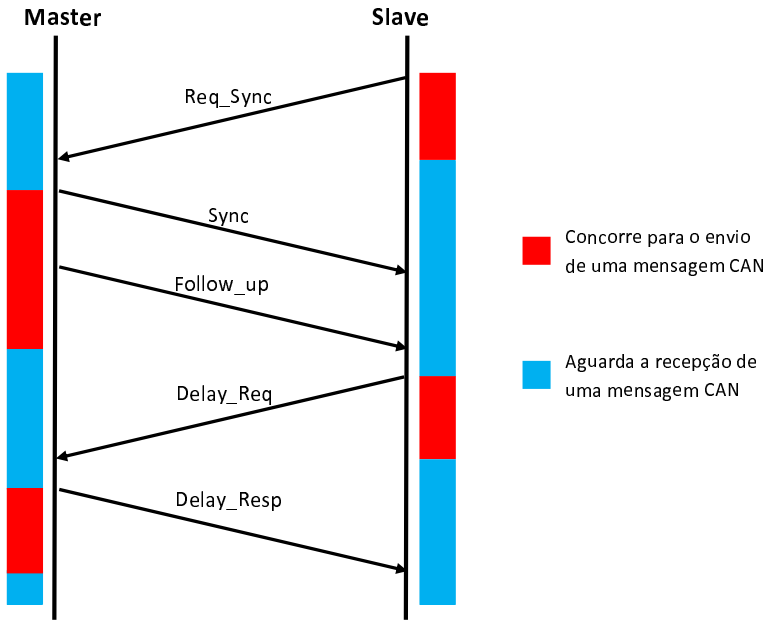


Figura 4.1: Esquema ilustrativo da concorrência entre envio de mensagens com o mesmo ID por parte do *Slave* e do *Master*.

4.2.4 *Timestamping*

O *timestamping* das mensagens de sincronização será feito no final da mensagem tanto na transmissão como na recepção de mensagens de sincronização, tal como ilustra a Figura 4.2.

Este acontecerá mais precisamente depois do envio/recepção do campo EOF da trama CAN (ver secção A.5.3).

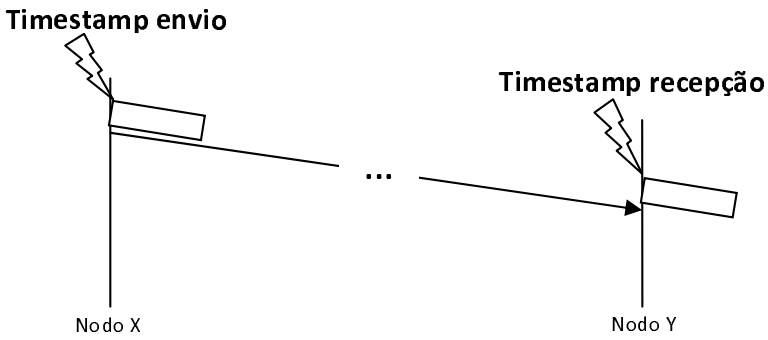


Figura 4.2: Forma como é feito o *timestamping* das mensagens de sincronização.

4.3 Plataforma de desenvolvimento

No âmbito da implementação do protocolo de sincronização em software, foram utilizados placas DETPIC, tal como mostra na Figura 4.3. A plataforma que foi disponibilizada contém sete destas placas ligadas entre si através de um barramento CAN. Foram utilizados microcontroladores da Microchip, nomeadamente o PIC18F258.

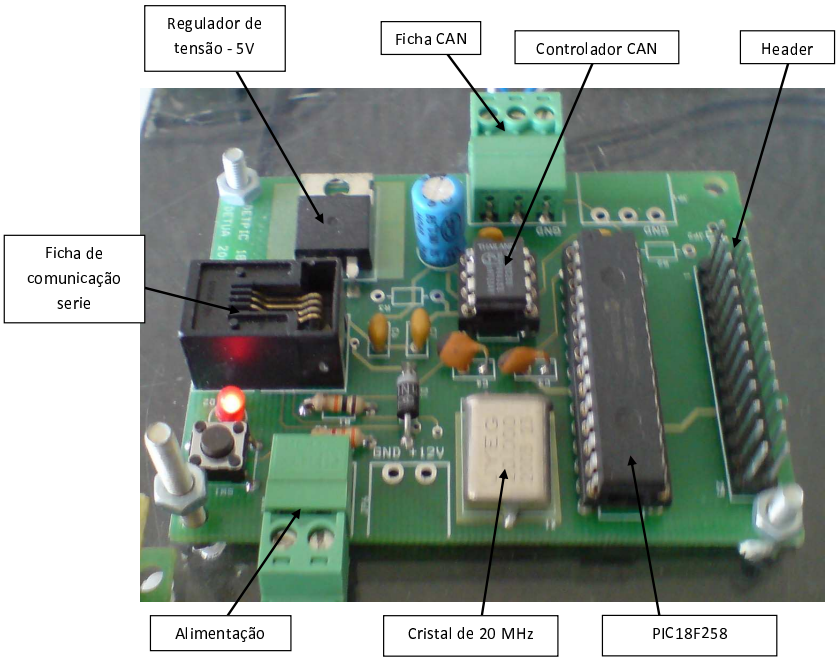


Figura 4.3: Placa DETPIC utilizada.

4.3.0.1 Características do microcontrolador

O PIC18F258 da Microchip disponibiliza:

. 3 portos I/O;

. 32Kbytes de RAM;

. 5 canais A/D;

. módulo de comunicação série;

. 4 timers;

. 17 fontes de interrupção;

. 2 níveis de prioridade nas interrupções.

Para mais detalhes ver em [Mic06].

4.4 RTKPIC18

Cada vez é mais comum a utilização de *Kerneis* no desenvolvimento de aplicações embutidas. Assim, no âmbito desta dissertação considerou-se a implementação efectuada quer directamente no PIC quer usando um *Kernel*. Este facto é relevante pois os *kerneis* têm tipicamente um impacto negativo na sincronização de relógio, devido a zonas de não preempção e secções criticas em que as interrupções se encontram desligadas.

O *Kernel* RTKPIC18 (*Real-Time Kernel* PIC18FXX8) foi desenvolvido na Universidade de Aveiro no Ano lectivo de 2003/2004. Este *kernel* de tempo real foi desenvolvido para microcontroladores da família PIC18FXX8 e escrito usando o PICC-18 da HI-TECH Software.

É um *kernel* multi-tarefa, preemptivo e com preocupações de tempo-real. Este permite definir até 13 tarefas periódicas, com relação de fase e deadline, que são escolhidas de forma transparente pelo utilizador. O *kernel* efectua escalonamento para prioridades fixas tal como *Rate Monotonic* (prioridade inversamente proporcional ao período) ou *Deadline Monotonic* (prioridade inversamente proporcional à *deadline* relativa). Já no escalonamento de prioridade dinâmicas este permite a utilização do *Earliest Deadline First* (prioridade proporcional à proximidade das *deadlines* em *runtime*).

O RTKPIC18 recorre ao *timer2* para a contagem de *ticks* e corre no nível alto de interrupções da PIC, pelo que se deverá ter cuidado em não utilizar estes recursos aquando da utilização do *kernel* [LM04].

4.5 Arquitectura

No protocolo implementado em software existem duas partes distintas, uma relativa à implementação do relógio *Master* e outra relativa à implementação do relógio *Slave*. Apesar de funcionalmente distintos, estes módulos apresentam diversas características em comum.

Os PIC18F258, possuem múltiplas fontes de interrupção que podem ser atribuídas a rotinas de alta ou de baixa prioridade. Os eventos da rotina de alta prioridade irão sobrepor-se a qualquer rotina de baixa prioridade que possa estar em execução.

Em ambas, a implementação do protocolo decorre ao nível da interrupção de baixa prioridade, sendo que a interrupção de alta prioridade é guardada para posteriormente utilizar o *kernel* RTKPIC18, a correr no microcontrolador simultaneamente com o protocolo de sincronização. Este servirá fundamentalmente para avaliar os efeitos nefastos, na sincronização de relógio, impostos pelos tempos de bloqueio do *kernel*. Estes tempos de bloqueio acrescidos irão naturalmente degradar a precisão na sincronização de relógio devido ao menor rigor conseguido na obtenção dos *timestamps*, que irão assim dar valores errados de *offset* e consequentemente resultarão em correcções erradas no relógio.

Ambos os componentes do protocolo podem ser tratadas como módulos independentes, permitindo a fácil integração em outros projectos. Para isso, e dado que a implementação dos relógios decorre ao nível das interrupções, desenvolveu-se um aplicativo que permite o registo de funções nas rotinas de interrupção, quer na de baixa prioridade quer na de alta prioridade. Isto apresenta-se como uma mais valia, visto que possibilita adicionar outras funções nas rotinas de interrupção de uma forma transparente, evitando-se ter de aceder ao código do módulo de sincronização de relógio ou mesmo do *kernel*. No anexos A.6 é explicado mais detalhadamente o funcionamento deste módulo bem como dos resultados de *overhead* obtidos.

4.5.1 Arquitectura do relógio *Slave*

A implementação deste relógio está assente na rotina de interrupção, tirando partido de três fontes de interrupção, nomeadamente da interrupção do *timer0* e da interrupção de recepção e transmissão do CAN para a obtenção de *timestamps*. Na Figura 4.4 é apresentado o diagrama de blocos básico relativo à implementação do relógio *Slave*.

Assim e seguindo o esquema da figura, ao ser gerada uma interrupção é efectuada a leitura do *timer0*, para assim evitar os atrasos inerentes à determinação da condição de *timestamp*, sendo depois este valor utilizado se necessário. Seguidamente:

(1) é verificada a ocorrência da interrupção do *timer0*, que caso se verifique levará à actualização do valor do tempo, sendo também verificada a necessidade de despoletar ou não novo processo de sincronização;

(2) caso ocorra ocorra uma interrupção da transmissão do CAN é guardado o valor de *timestamp* sendo para isso necessário somar o valor anteriormente lido do *timer0* em microsegundos com o valor actual do tempo (necessário no envio da mensagem *Delay_Req*);

(3) é verificada a ocorrência de uma interrupção de recepção do CAN, que caso se verifique leva a efectuar o *timestamp* tal como anteriormente (necessário na recepção da mensagem *Sync*). Mediante o estado de sincronização pode ser necessário executar a lógica e cálculos necessários durante o processo de sincronização, nomeadamente os cálculos de *offsets* e o ajuste do relógio;

(4) procede-se ao envio de mensagens de sincronização se houver necessidade disso.

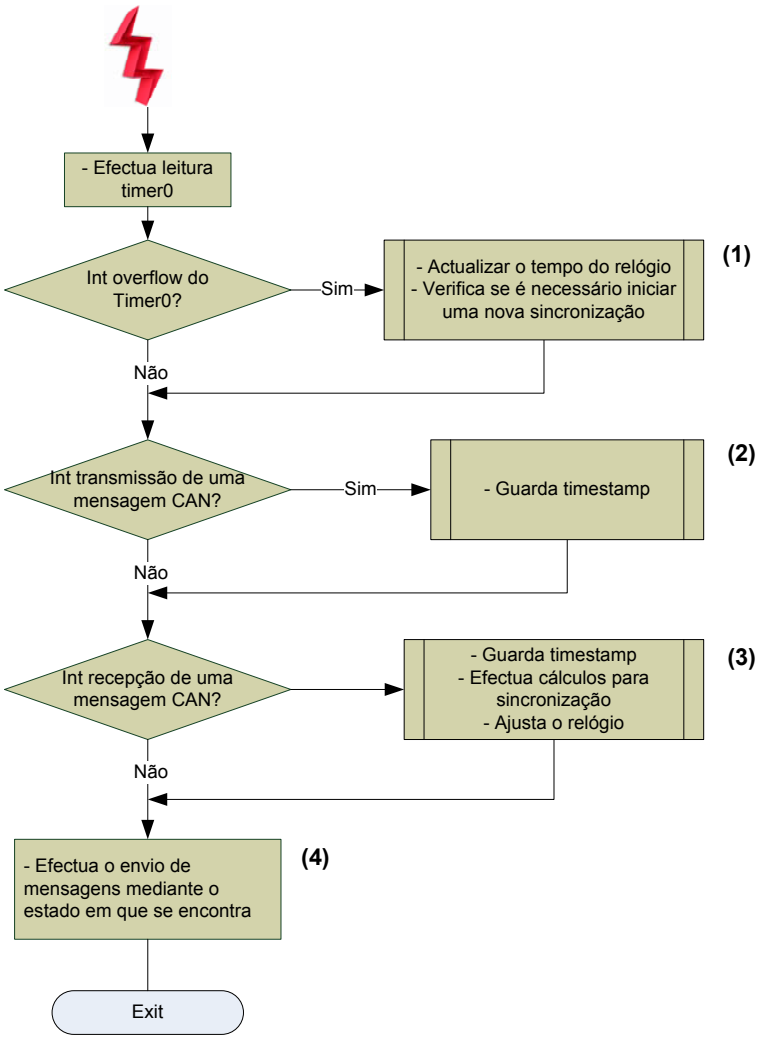


Figura 4.4: Diagrama de blocos da implementação do relógio *Slave*.

4.5.2 Arquitectura do relógio *Master*

Tal como no relógio *Slave*, o código associado ao relógio *Master* encontra-se na rotina de interrupção (baixa prioridade) tirando partido das mesmas fontes de interrupção, como se pode ver na Figura 4.5.

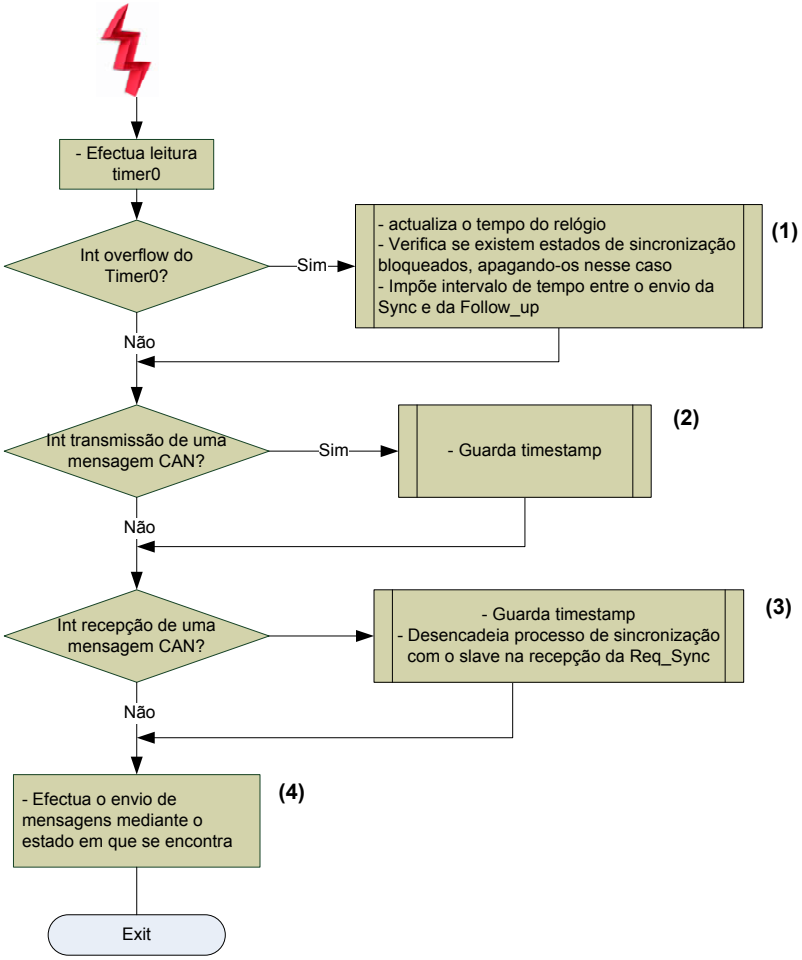


Figura 4.5: Diagrama de blocos da implementação do relógio *Master*.

Apesar de não se encontrar representado no esquema, o relógio *Master* possui um *array* de estruturas (explicados mais em detalhe na secção 4.6.4) que permite a sincronização de vários *Slaves* em simultâneo. Assim em cada etapa será necessário percorrer o *array* de estruturas a fim de verificar se nessa etapa alguma das sincronizações tem procedimentos pendentes que devam ser realizados.

Seguindo agora o esquema da figura, ao ser gerada uma interrupção é efectuada a leitura do *timer0*, evitando assim os atrasos de o fazer só na chegada à condição de *timestamp*, sendo aí utilizado se necessário. Seguidamente:

(1) é verificada a ocorrência da interrupção do *timer0*, que caso se verifique, leva à actualização do valor do tempo. Também é verificada a existência de estados de sincronização com *Slaves* que não se alterem durante um determinado tempo (tipicamente 1 segundo).

Caso isso aconteça será limpa a estrutura de sincronização relativamente aos *Slaves* em que a condição se verifique, sendo o procedimento de sincronização com esses *Slaves* abortado. Por fim, se o *Master* tiver enviado uma mensagem *Sync* para um *Slave* força aqui a um estado de espera (*wait*), para só a seguir passar ao envio da mensagem *Follow_up*;

(2) caso se verifique a interrupção de transmissão do CAN, levará a guardar o valor de *timestamp*, sendo para isso necessário somar o valor do *timer0* anteriormente capturado (em microsegundos) com o valor actual do tempo (necessário no envio da mensagem *Sync*). Este valor será guardado no campo da estrutura do *Slave* para o qual a mensagem foi enviada.

(3) se ocorrer a interrupção de recepção do CAN é guardado o *timestamp* tal como anteriormente (necessário na recepção da mensagem *Delay_Req*), bem como é desencadeado o processo de sincronização com um *Slave* mediante a recepção do pedido de sincronização.

(4) procede-se ao envio de mensagens de sincronização se houver necessidade disso.

4.6 Alguns aspectos de implementação

Ao nível da implementação dos relógios a estratégia usada foi baseada em duas máquinas de estados ilustrada na Figura 4.6, em que cada estado representa uma etapa do protocolo de sincronização. Assim e começando pelo *Master*:

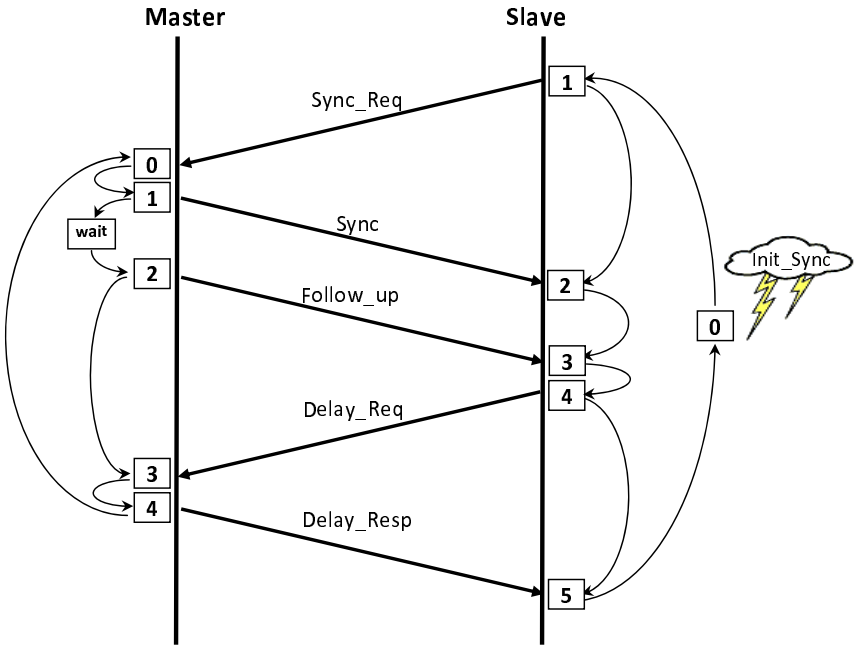


Figura 4.6: Esquema relativo aos estados presentes em cada etapa durante o processo de sincronização no relógio *Master* e *Slave*.

0 - Espera pela recepção de um pedido de sincronização por parte do *Slave*;

1 - Procede ao envio da mensagem *Sync* para o *Slave*;

2 - Depois de passar um intervalo de tempo predeterminado, para evitar picos de processamento no *Slave*, é enviada a mensagem *Follow_up*;

3 - Espera pela recepção da mensagem *Delay_Req*;

4 - Envia a mensagem *Delay_Resp*.

O *Slave* passa pelos estados:

0 - Estado desactivo do protocolo até ser despoletada o início de uma nova sincronização;

1 - Procede ao envio da mensagem *Sync_Req*;

2 - Espera pela recepção da mensagem *Sync*;

3 - Espera pela recepção da mensagem *Follow_up*, e após a sua chegada procede ao cálculo do *offset*;

4 - Envia a mensagem *Delay_Req*;

5 - Aguarda a recepção da mensagem *Delay_Resp*, e depois da sua chegada é feito o cálculo do atraso de propagação, bem como corrigido o relógio.

4.6.1 Implementação do relógio

O relógio do sistema é composto por uma variável mantida em software e pelo valor presente no *timer0*. A variável mantida em software é incrementada pela interrupção do *timer0* de 13 em 13 ms. O intervalo entre sincronizações é ajustável por meio de variações do valor com que o contador associado ao *timer0* é carregado (*Start_timer0*).

Estabeleceu-se que seria desejável que este relógio funcionasse durante pelo menos um ano sem que ocorresse o *wrap around*. Se este tiver uma resolução de 1 microsegundo, são necessários pelo menos 45 bits para a variável de relógio, dado que 1 segundo tem 10^6 microsegundos, 1 minuto tem 60 segundos, 1 hora tem 60 minutos, 1 dia tem 24 horas, e um ano por excesso tem 366 dias, assim:

$$1 \times 10^6 \times 60 \times 60 \times 24 \times 366 \approx 3.17 \times 10^{13} \tag{4.5}$$

$$2^n = 3.17 \times 10^{13} \Rightarrow n = \frac{\log(3.17 \times 10^{13})}{\log(2)} \approx 44.8 \Rightarrow n = 45bits \tag{4.6}$$

Assim a variável de relógio é representada por um *unsigned int* para os bits menos significativos e um *unsigned long* para os restantes bits, perfazendo assim um total de 16 + 32 = 48 bits. Estes irão permitir que o *wrap around* só se verifique passados 8.9 anos. Nesta abordagem sempre que se adicionar a passagem do tempo na variável, tem de se verificar a ocorrência de *overflow* da variável *unsigned int* para a correcta manutenção da variável de relógio, tal como se pode ver na Figura 4.7.

```
//incrementa tempo
if((0xFFFF - abs_time_low) >= Clock_Increment)
{
        abs_time_low +=Clock_Increment;

}
else
{
        abs_time_low = Clock_Increment - (unsigned int)( 0x00010000 - (unsigned long)abs_time_low);
        abs_time_high = abs_time_high + 1;
}
```

Figura 4.7: Forma como é feito o incremento de tempo.

4.6.1.1 Leitura do Relógio

Para se fazer uma leitura do tempo actual do relógio é necessário adicionar à variável de relógio mantida em software, os microsegundos decorridos no *timer0* desde a última interrupção por ele gerada, como ilustra o diagrama de blocos da Figura 4.8.

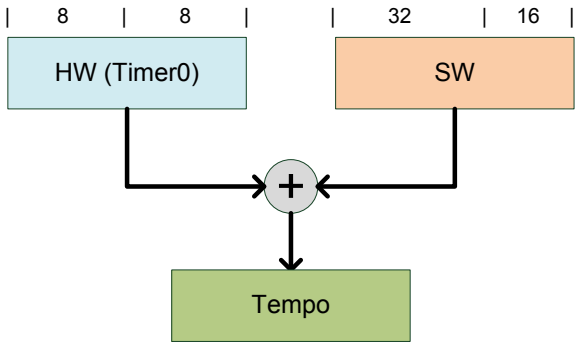


Figura 4.8: Forma básica como é obtido o tempo do relógio.

4.6.2 Actuação no relógio

Existem dois modos de actuação no relógio. Um deles consiste em corrigir o relógio ao nível da variável de tempo, no decorrer do processo de sincronização. Neste é corrigido primeiramente o desvio causado pelo *offset* e depois corrigido o atraso de propagação, tal como vem explicado na secção 2.6 do Capítulo 2 e como se pode ver na Figura 4.9. Este método de correcção servirá essencialmente para fases de inicialização em que os relógios podem estar muito desfasados, permitindo assim um ajuste rápido para que, a partir daí se utilize uma correcção ao nível de hardware que permita ter um relógio monotónico.

O outro modo de correcção é baseado na actuação ao nível do hardware, nomeadamente no ajuste do valor de início de contagem do *timer0* após a sua interrupção (*Start_Timer0*). Utilizando um compensador PID que a partir do resultado global de *offset*, ou seja, do desvio causado pelo *offset* mais o atraso de propagação, irá determinar qual o *Start_timer0* que permita diminuir o *offset*. Além de permitir correcções suaves, este método também permite a manutenção de um relógio monotónico, necessário em inúmeras aplicações. Em relógios monotónicos garante-se que uma leitura de tempo subsequente a outra terá sempre um valor igual

ou superior relativamente à leitura anterior. Isto é de extrema importância em transacções financeiras, entre outras aplicações.

Na Figura 4.9 é possível ver os dois métodos de actuação no relógio e como é que é feita a correcção do mesmo.

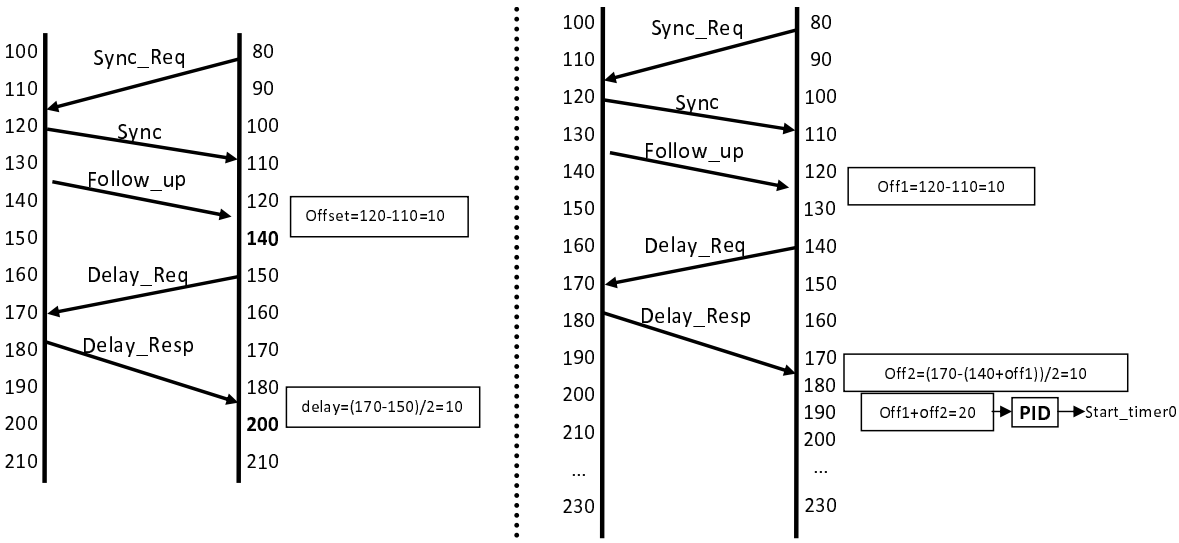


Figura 4.9: Duas formas distintas de corrigir o relógio. Do lado esquerdo o relógio é corrigido à medida que os *offsets* vão sendo calculados, e do lado direito depois de calculados os *offsets* é feita uma actuação no relógio através do *Start_timer0*.

4.6.3 Implementação do relógio *Slave*

4.6.3.1 Cálculos decorrentes do protocolo

Além dos cálculos necessários para efectuar a manutenção do relógio no *Slave* e já explicados na secção 4.6.1, existem vários cálculos a efectuar durante o processo de sincronização e que necessitam de cuidados especiais. Estes cuidados vêm sobretudo dos cálculos terem de ser efectuados com registos de 48 bits (32+16), resultantes da variável de relógio ser composta por um *unsigned int* para os bits menos significativos e por um *unsigned long* para os restantes bits, bem como o facto de os resultados intermédios com estes registos muitas vezes poderem dar resultados negativos.

Assim sendo, o procedimento utilizado consistiu em trabalhar sempre com resultados positivos tendo associada uma *flag* que indica se o resultado é positivo ou negativo. Assim em contas subsequentes, essa *flag* permitirá decidir se devemos somar ou subtrair o valor. Se por um lado esta abordagem permite evitar algumas complicações decorrentes de obter e tratar números negativos com 48 bits num PIC, por outro aumenta o número de condições necessárias para fazer uma conta.

É de esperar que num microcontrolador de baixa performance, todos estes cálculos decorrentes do processo de sincronização, tenham um impacto não negligenciável na carga de processamento do PIC e nos tempos de bloqueios impostos para realização dos cálculos.

4.6.4 Implementação do relógio *Master*

4.6.4.1 *Array* de estruturas

No intuito de permitir que o relógio *Master* sincronize vários *Slaves* em simultâneo, houve necessidade de implementar um *array* de estruturas em que cada posição seria ocupada com dados relativos à sincronização de um relógio *Slave*. Além disso, foi necessário alterar o algoritmo do *Master* de forma a que para cada tipo de evento, este percorresse os vários *arrays* a fim de verificar a necessidade ou não de realizar o evento nas sincronizações pendentes.

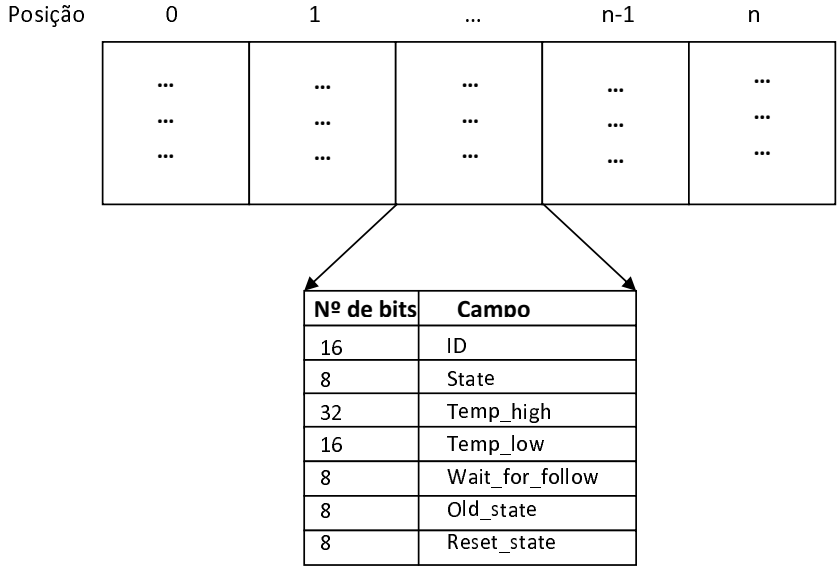


Figura 4.10: Definição do *array* de estruturas com os respectivos campos necessários em cada posição.

Na Figura 4.10 é representado o *array* de estruturas utilizado, bem como o tamanho e nome dos campos contidos em cada posição do *array*. Assim no campo *ID* fica guardado o ID do *Slave*, nos campos *temp_high* e *temp_low* é guardado um valor timestamp. O *wait_for_follow* contém um contador que é incrementado, após o envio da mensagem *Sync*, até um determinado valor e que permite fazer um tempo de espera entre o envio da mensagem *Sync* e *Follow_up*.

Finalmente, os campos *old_state* e *reset_state* são necessários no mecanismo de verificação de estados bloqueados até 1 segundo. Após este tempo será feita a limpeza de todos os campos dessa posição do *array*, eliminando desta forma a sincronização pendente. Assim o campo *old_state* guarda o último estado registado e o *reset_state* é um contador que é incrementado sempre que a condição de igualdade entre o estado actual e o estado anterior se verifique. O contador será inicializado sempre que a condição não se verifique.

4.7 Conclusão

Neste capítulo foram abordadas as adaptações efectuadas ao protocolo IEEE 1588, passando este a designar-se de 1588Light. Estas alterações vieram sobretudo do facto da implementação realizada ser para aplicar em microcontroladores de baixa performance e com

largura de banda limitada. Assim, optou-se por simplificar o conteúdo das mensagens do PTP, de forma a estas não ocuparem mais do que um trama CAN, bem como se partiu do princípio que a rede já tinha um relógio *Master* predefinido, tornando-se desta forma o algoritmo de sincronização significativamente mais leve.

Seguidamente foram apresentadas as placas de desenvolvimento usadas na implementação em software (DETPIC), cujos controladores são PIC18F258 da Microchip. Foi explicada a implementação do 1588Light em software, bem como as suas principais características.

Capítulo 5

Implementação em Hardware

5.1 Introdução

A evolução da tecnologia microelectrónica ao longo das últimas décadas tem permitido um impressionante aumento da capacidade lógica dos circuitos integrados, sendo actualmente possível a construção de circuitos digitais complexos, específicos ou programáveis e integrados numa única pastilha.

Antes do aparecimento da lógica programável, os circuitos digitais eram construídos em placas de circuitos impresso utilizando componentes padrão, ou seja, eram usadas portas lógicas e blocos básicos (ex. somadores, *multiplexers*, *flip-flops*, contadores, etc) de diferentes circuitos integrados em que cada qual tinha uma função específica.

Com o aumento da complexidade dos componentes lógicos a utilizar e dos próprios sistemas a desenvolver sugiram circuitos integrados que permitem realizar quaisquer blocos lógicos. Estes circuitos integrados podem ser ASICs (*Application Specific Integrated Circuit*) ou FPGAs (*Field Programmable Gate Arrays*). Os ASICs apesar do melhor desempenho em relação às FPGAs, não são reconfiguráveis, sendo que a sua prototipagem e fabricação representam um elevado custo para projectos que não necessitem da produção de alguns milhares de unidades.

As FPGAs são matrizes de blocos lógicos programáveis conectados por recursos de interligação também programáveis. Por sua vez, cada bloco lógico programável pode ser constituído por vários elementos tais como tabelas de verdade, multiplexadores, portas lógicas, *flip-flops*, etc [dO07].

As FPGAs contêm inúmeras portas lógicas idênticas, também designadas de *LookUp Table* (LUT). As LUT, com N entradas (tipicamente entre 4 e 6) e uma saída, permitem implementar qualquer função booleana de N entradas. Estas portas lógicas padrão podem ser configuradas de forma independente e serem interconectadas a partir de uma matriz de fios condutores e *switches* programáveis.

A configuração destes elementos é conseguida pela programação das células SRAM presentes na FPGA, que guardam a configuração desses elementos sempre que programadas. Se a alimentação for desligada ou for feito *reset* à FPGA o conteúdo das memórias SRAM será limpo, podendo esta de futuro voltar a ser novamente programada.

A partir de ferramentas de projecto assistido por computador (CAD/CAE), tal como o ISE da Xilinx, são gerados arquivos binários para configuração da FPGA, através captura de esquemático, diagramas de transição ou da linguagem de descrição de hardware VHDL ou

mesmo através de linguagens de modelação ao nível do sistema (ex. *System Generator*). Este arquivo binário contém as informações necessárias para especificar a função de cada unidade lógica e para selectivamente fechar os *switches* da matriz de interconexão, permitindo assim ao utilizador construir circuitos integrados complexos.

Dado que os arquivos binários são, depois de transferidas, carregados em memórias RAM na FPGA, é possível serem reconfiguradas com novos arquivos binários sempre que necessário, oferecendo assim grande reconfigurabilidade às FPGAs.

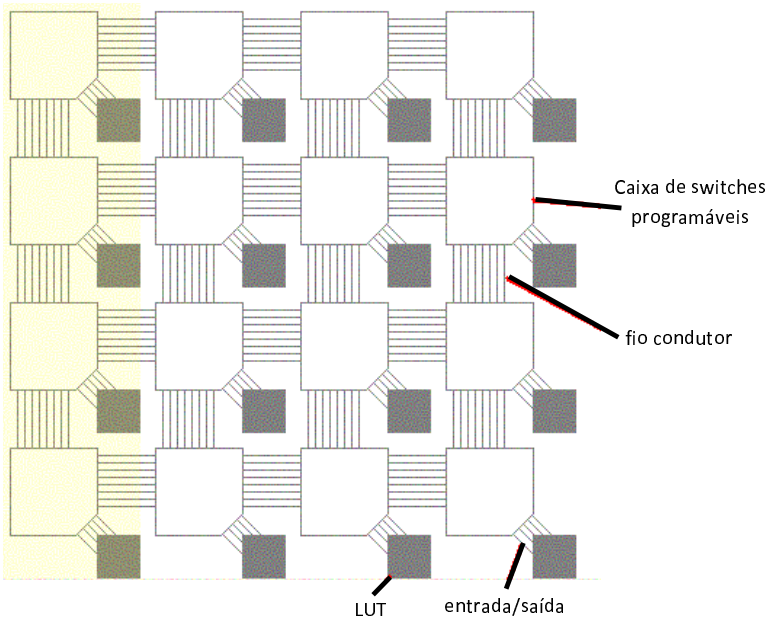


Figura 5.1: Arquitectura interna de uma FPGA.

As FPGAS são bastante poderosas, relativamente baratas, e bastante adaptáveis. As vantagens e limitações das FPGAs dependem do tipo de aplicações alvo, da flexibilidade, do tempo disponível para o projecto do sistema e do número de unidades a construir. Estas possuem inúmeras vantagens em implementações de multiprocessadores [JS05], processadores multi-tarefa [dO07], processamento de sinal [DP06], telecomunicações [SBM06], coprocessador pra aplicações específicas [BVAK06], entre outros. A utilização de FPGAs pode também ser interessante em aplicações de sincronização de relógio apertada. A mais valia destas vem sobretudo da precisão com que se consegue obter os *timestamps*, sendo a qualidade desta medição fundamental para uma sincronização de relógio com valores de precisão e exactidão bastante melhores do que os conseguidos em aplicações realizadas somente em software. Outro aspecto resulta da capacidade das FPGAs integrarem simultaneamente diversos módulos do sistema.

Assim neste capítulo será abordada a implementação do protocolo 1588Light (explicado no capítulo anterior) em CAN numa solução baseada em hardware, sendo para isso utilizada a placa de desenvolvimento RC10 da Celoxica contendo uma FPGA Spartan-3. O módulo realizado englobará um controlador CAN (CLAN), a implementação do protocolo 1588Light bem como um mecanismo responsável por gerir a recepção e o envio de mensagens do controlador CAN.

5.2 Arquitectura

A arquitectura implementada, tal como é mostrado na figura 5.2 é constituída por 3 grandes blocos. Esta foi baptizada de CAN1588Light, dado que possui um controlador CAN, bem como um sistema de relógio sincronizado baseado no IEEE 1588. Os 3 grandes blocos desta implementação são o CLAN, o módulo de sincronização (1588Light) e o multiplexador de mensagens CAN.

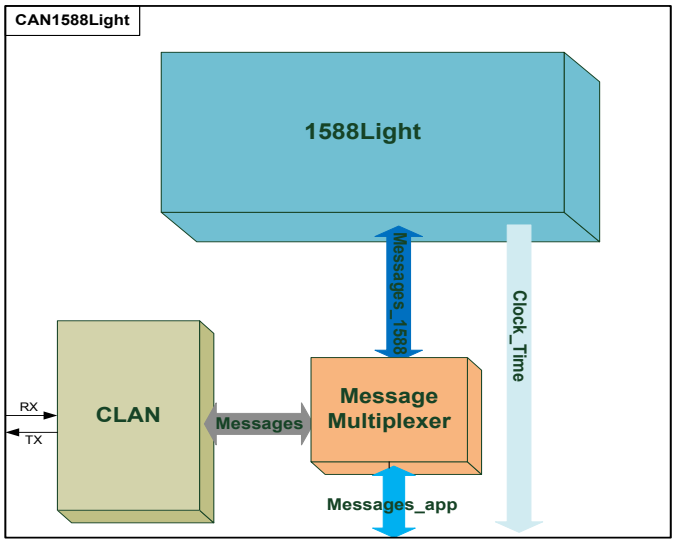


Figura 5.2: Os três grandes blocos da arquitectura do CAN1588Light.

5.2.1 CLAN

O CLAN é um controlador CAN que foi desenvolvido de raiz no Departamento Electrónica, Telecomunicações e Informática da Universidade de Aveiro [OAF05]. Este foi disponibilizado na forma de núcleo de propriedade intelectual modelado em VHDL, sintetizável e implementável em FPGAs da Xilinx.

O núcleo CLAN implementa a especificação CAN 2.0B, funcionando como camada física e MAC do protocolo CAN. Este permitirá a transmissão e a recepção de mensagens CAN no barramento, bem como fornece sinais de controlo e de estado, que entre outras coisas, irão permitir a obtenção dos instantes de recepção e envio de mensagens CAN, necessários no processo de sincronização.

5.2.2 *Message Multiplexer*

O multiplexador de mensagens serve, como o próprio nome diz, para multiplexar as mensagens entre o bloco de sincronização e a aplicação que utilize o módulo CAN1588Light. Assim é remetido para este bloco a gestão da concorrência nos pedidos de envio de mensagens, assim como a resolução do encaminhamento de mensagens recebidas, sendo que as mensagens do protocolo de sincronização não passarão para a aplicação, bem como as mensagens da aplicação não serão recebidas no bloco de sincronização de relógio.

Assim o utilizador do módulo 1588Light, irá vê-lo como uma caixa preta, não tendo necessidade de se preocupar com o facto de o módulo de sincronização também utilizar o barramento CAN. Tudo o que o utilizador da aplicação precisa de fazer quando necessita de enviar uma mensagem CAN, é ter o sinal de pedido de envio mensagem activo até que receba um sinal a informar que a mensagem foi enviada. Todo o restante processo é realizado dentro do bloco *Message Multiplexer* através da utilização de uma máquina de estados.

5.2.3 1588Light

O módulo 1588Light é responsável pela manutenção de um relógio, que estará sincronizado com os restantes relógios da rede CAN. Para manter esta coerência temporal entre os relógios da rede, o módulo implementa o protocolo 1588Light (já descrito em 4.2) por forma a periodicamente sincronizar os relógios.

Assim serão necessárias realizar duas implementações para este módulo, uma para os nodos *Slave* e outra para o nodo *Master*.

5.2.3.1 Implementação do relógio *Slave*

Na Figura 5.3 é apresentado o esquema básico da arquitectura implementada para um relógio *Slave* (1588Light - *Slave*). Este módulo é constituído por 5 grandes blocos, são eles o *Init_Sync*, o *State_Machine*, o *Actuator*, o *Cnt_utick* e o *Clock*.

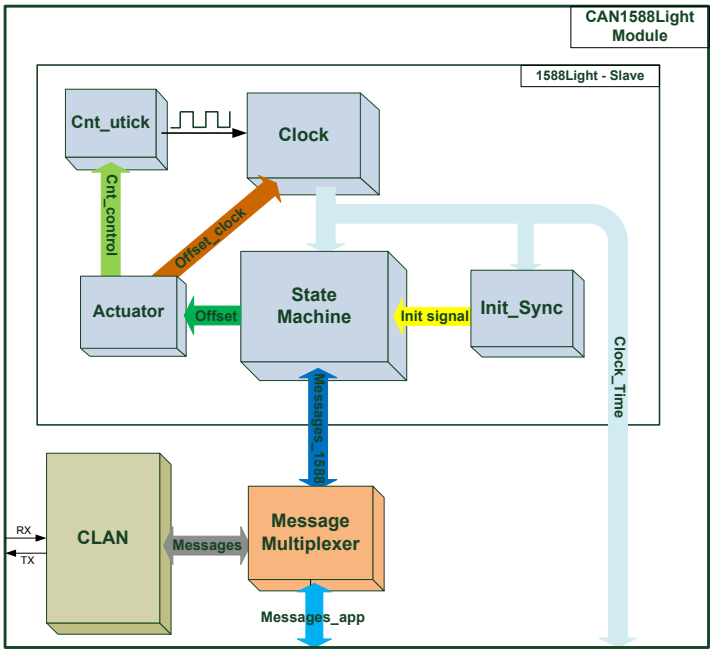


Figura 5.3: Arquitectura do módulo CAN1588Light para um relógio *Slave*.

Seguidamente será explicado cada um destes blocos.

Init_Sync

O bloco *Init_Sync* é responsável por desencadear o processo de sincronização na *State_Machine* com um determinado período, que pode ser definido pelo utilizador, sendo que os valores tipicamente utilizados são 1, 2, 8 ou 64 segundos.

State_Machine

O bloco *State_Machine* está encarregue de todo o processo decorrente do protocolo de sincronização. Este procede ao envio e recepção das mensagens PTP bem como captura os *timestamps* e calcula o *offset* de relógio em relação ao relógio do *Master*.

O protocolo de sincronização é implementado através de uma máquina de estados, sendo que cada estado representa uma etapa do protocolo de sincronização, seguindo assim a mesma abordagem utilizada na implementação em software (ver em 4.6).

Actuator

O bloco *Actuator* é responsável por corrigir o relógio. Este mediante o valor de *offset* obtido pela *State_Machine* actua em conformidade. Existem duas formas possíveis de corrigir o relógio, sendo que o factor de decisão, pelo método a utilizar, é o valor absoluto do *offset*.

Para valores pequenos de *offset*, o *Actuator* actuará ao nível da velocidade de contagem do relógio, ajustando o valor de *cnt_utick*. Este será obtido a partir de um compensador PID (Proporcional-Integral-Derivativo) com o valor de *offset* como entrada.

Caso o *offset* ultrapasse um determinado valor é feita uma correcção imediata do valor do relógio. Esta será feita passando o valor de *offset* para o sinal *Offset_clock*, que forçará o relógio a ajustar o seu valor de forma a ficar sincronizado.

Cnt_utick

O bloco *Cnt_utick* possui um algoritmo que permite ajustar, de forma fina, a velocidade de contagem do relógio. Este mediante o valor do sinal *cnt_utick*, estabelece a frequência com que activa o sinal responsável por provocar o incremento de tempo no relógio.

Clock

O bloco *Clock* é o relógio do sistema. Este mediante o sinal fornecido pelo bloco *Cnt_utick* incrementa o relógio, tomando em atenção o valor do sinal *Offset_clock* sempre que necessário efectuar correcções do valor de relógio.

5.2.3.2 Implementação relógio *Master*

Na Figura 5.3 é apresentado o esquema básico da arquitectura implementada para um relógio *Master* (1588Light - *Master*). Este módulo é constituído 5 blocos diferentes, são eles o *Master Cell*, o *Cell Activator*, o *Transmission Multiplexer*, o *Cnt_utick* e o *Clock*.

Seguidamente será brevemente explicado cada um destes blocos.

Master Cell

De forma a permitir que vários *Slaves* sincronizem em simultâneo num mesmo *Master*, este módulo de sincronização possui um número de blocos *Master Cell* que é parametrizável.

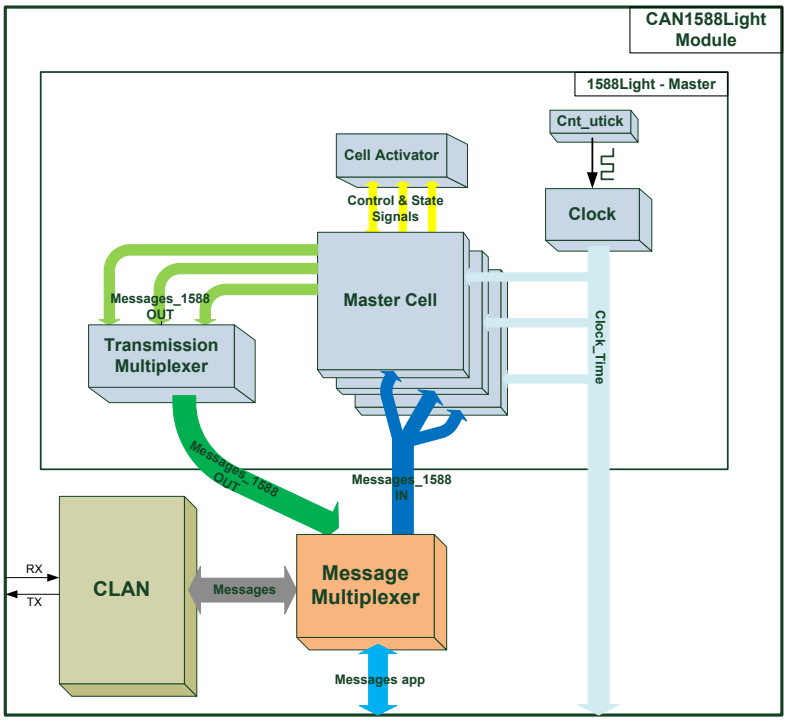


Figura 5.4: Arquitectura do módulo CAN1588Light para um relógio *Master*.

Cada bloco destes é responsável pelo processo de sincronização com um *Slave*, sendo que a gestão da activação dos blocos para sincronizarem com os vários *Slaves* é efectuado no bloco *Cell Activator* através da troca de sinais de controlo.

O bloco *Master Cell* é implementado com base numa máquina de estados, seguindo a mesma filosofia utilizada no relógio *Slave*. Este bloco possui ainda um mecanismo que permite detectar situações em que a sincronização com o *Slave* tenha ficado bloqueada por falha ou interrupção. Nestas situações o bloco elimina a sincronização pendente ficando livre para sincronizar com outros *Slaves*.

Cell Activator

O bloco *Cell Activator* permite gerir a utilização de blocos *Master Cell*, activando estes blocos mediante as mensagens PTP recebidas. Este além de gerir a utilização dos blocos é responsável por sinalizar ao bloco correcto a recepção de mensagens de sincronização, bem como limpar o *buffer* de recepção do controlador CAN após o *Master Cell* seleccionado ter recebido a mensagem.

Transmission Multiplexer

O bloco *Transmission Multiplexer* está encarregue de gerir os pedidos de envio de mensagens CAN pelos blocos *Master Cell*, sejam eles concorrentes ou não.

A filosofia utilizada é muito parecida à do bloco *Message Multiplexer*, no entanto o *Transmission Multiplexer* apenas trata a concorrência da transmissão de mensagens

CAN. Este utiliza um sistema baseado em máquina de estados de forma a gerir os pedidos por parte dos vários blocos *Master Cell*.

Cnt_utick

O bloco *Cnt_utick* força o relógio a funcionar a uma frequência fixa. Esta não será passível de ser ajustada, dado que este relógio (*Master*) se apresenta como a base de tempo correcto.

Clock

O bloco *Clock* é o relógio do sistema. Este mediante o sinal fornecido pelo bloco *Cnt_utick* incrementa o relógio. Este não permite a alteração brusca do seu valor, tal como acontecia no *Slave*.

5.3 Alguns aspectos de implementação

5.3.1 Algoritmo de correcção do relógio

Um factor importante para manter uma sincronização de relógio exacta está relacionada com o grau de ajuste que se consegue efectuar na taxa de contagem do relógio. Quanto menor for o grau de ajuste, mais precisas e mais suaves serão as correcções possíveis efectuar no relógio.

O bloco *Cnt_utick* tal como estava desenhado para o *Master* só permitia que os ajustes fossem feitos ao nível do valor de final de contagem. Este ao ser mudado para a unidade seguinte (ou anterior) provocaria um ajuste muito grosseiro. Esta mudança representaria um acréscimo (ou decréscimo) de 20.83ns em cada contagem microsegundo. Sendo que 1 segundo tem 1 milhão de microsegundos, o ajuste mínimo conseguido seria da ordem de

$(20.8ns * 1 \times 10^6) \times 2 = 41.67ms$ se as sincronizações ocorressem de 2 em 2 segundos.

Uma possibilidade seria aumentar a frequência de funcionamento deste bloco. Mas mesmo aumentando esta cerca 4 vezes (f≈200MHz) não seria possível melhorar significativamente este valor. Assim optou-se por desenvolver um novo bloco *Cnt_utick*. Este novo bloco irá permitir realizar ajustes muito mais finos no relógio do *Slave*, sendo o grau de ajuste teórico possível da ordem dos 42ns entre intervalos de sincronização.

Nesta implementação será utilizado um algoritmo que através do número de contagens de microsegundos existentes entre sincronizações, de uma parte inteira e fraccionaria resultante da compensador PID (no bloco *Actuator*) permitirá ir ajustando o valor de final de contagem no contador.

Por exemplo, se o número de contagens entre sincronizações (*Ncounts*) for 10, a parte inteira (*Cnt_int*) é igual a 2 e a parte fraccionaria (*Cnt_frac*) é igual a 2, o algoritmo tem de conseguir que nas 10 contagens, duas delas sejam com o valor 3 e as outras 8 serem com o valor 2. Além disso, este irá distribuir os diferentes valores de contagens de forma uniforme entre duas sincronizações.

Este mesmo exemplo pode ser visto na Figura 5.5. A risca azul representa as contagens para o valor 2 e a risca vermelha representa as contagens para o valor 3. No gráfico verifica-se também que o relógio *Slave*, mesmo sendo um pouco mais rápido que o *Master*, consegue ter o seu relógio sincronizado (10μs no *Slave* = 10μs no *Master*).

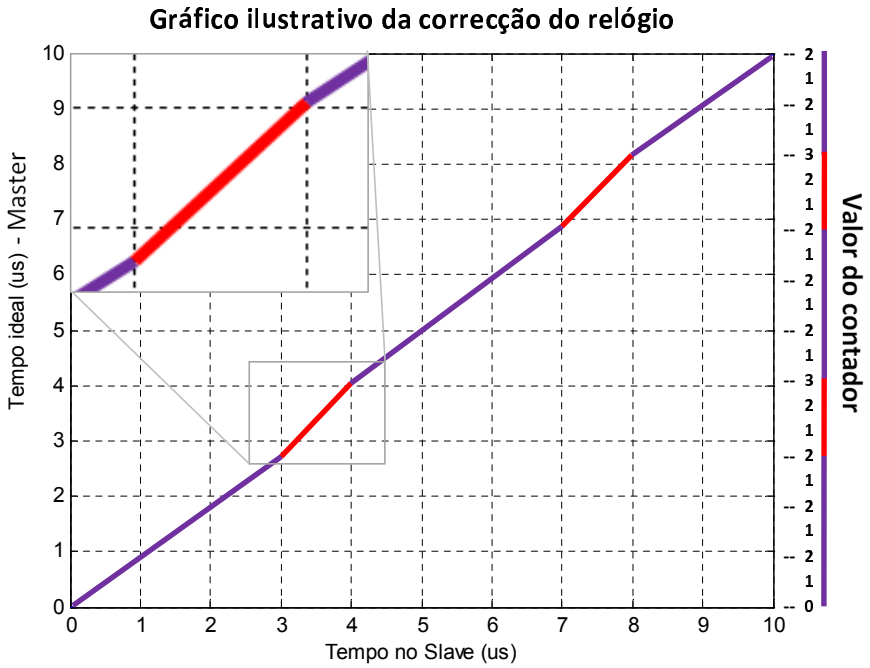


Figura 5.5: Forma como é ajustada a velocidade de contagem do relógio *Slave* de forma a ficar sincronizado com o relógio *Master*.

Na Figura 5.6 é apresentado o esquema deste novo bloco *Cnt_utick* para o relógio *Slave*.

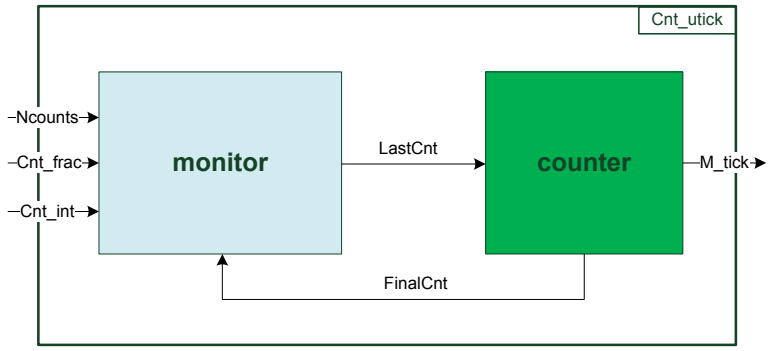


Figura 5.6: Esquema do bloco Cnt_utick no relógio *Slave*.

Este é constituído por dois blocos principais, são eles o *counter* e o *monitor*. O *counter* é um contador que sempre que chegar ao valor do sinal *LastCnt* recomeça a contagem forçando o relógio do sistema (o bloco *Clock*) através do sinal *M_tick* a incrementar 1 microsegundo na sua variável de relógio. Além disso, no final de contagem através do sinal *FinalCnt*, o bloco *counter* induz o bloco *monitor* a determinar um novo valor para o *LastCnt*.

O bloco *monitor* é o responsável por determinar os valores do *LastCnt* ao longo do tempo. Esta decisão é baseada nos sinais *Ncounts* (número total de contagens de microsegundos entre sincronizações) e nos sinais *Cnt_frac* (parte fraccionaria do valor de *Cnt*) e *Cnt_int* (parte

inteira do valor *Cnt*). O valor *Cnt* é o sinal de saída do compensador PID no bloco *Actuator*.

5.4 Plataforma de desenvolvimento

A placa base de desenvolvimento da implementação em hardware foi a RC10 da Celoxica contendo uma FPGA Spartan-3. Esta FPGA contém cerca de 1.5 milhões de portas lógicas.

Seguidamente são apresentados os principais elementos da placa de desenvolvimento utilizados directamente ou indirectamente para realizar as implementações em hardware. Será também referido de forma sucinta o motivo da sua utilização.

Módulo de Sincronização (CAN1588Light)

1. Xilinx Spartan 3L XC3S1500L-4-FG320.
   Utilizada para executar em hardware as implementações realizadas (*Master* e *Slave*). Também foram utilizadas para implementar alguns mecanismos de depuração e monitorização dos sistemas.
2. Conector para barramento CAN.
   Permitiu a ligação física do controlador CAN ao barramento.

***Debug*, monitorização das implementações e recolha de resultados**

1. *Joystick* com 5 posições.
   Permite ajustar em "tempo-real"os valores das constantes (KP, KI e KD) do compensador PID, funcionando muito semelhante ao *joystick* de um telemóvel.
2. Porta série (RS-232)
   Utilizada para enviar para o PC informação periódica de sinais internos da implementação, tais como, tempo do relógio, *offset* valores das constantes do PID, entre outros. Estes são tratados e mostrados através da ferramenta *Matlab*.
3. Dois *displays* de sete segmentos
   Permitiu a visualização de sinais internos do sistema.
4. Oito LEDs verdes
   Para monitorizar estados do processo de sincronização nas implementações, bem como outro tipo de sinais relevantes.
5. 50 pin *expansion header*
   Utilizado no intuito de colocar sinais de *timestamp* fora da FPGA para depois serem utilizados pelo medidor de *offset*.
6. Conector JTAG
   Utilizado inicialmente para fazer a depuração das implementações (utilizando o ChipScope).

5.5 Recursos utilizados na FPGA

Os recursos da FPGA usados na implementação são bastante reduzidos, prevendo-se assim uma fácil integração do módulo CAN1588Light em outros projectos na mesma FPGA.

5.5.1 Aplicação *Master*

Os principais recursos utilizados da FPGA na implementação do do módulo CAN1588Light(Master) foram:.

```
Final Synthesis Report
=================================================
Selected Device :xc3s1500l-4fg320
Device utilization summary:
N? of Slices: 1538 out of 13312 (11%)
N? of Slice Flip-Flops: 1292 out of 26624 (4%)
N? of 4 input LUTs: 2280 out of 26624 (8%)
N? of Bonded IOBs: 11 out of 221 (4%)
N? of Global CLKs: 2 out of 8 (25%)
N? of Block RAMs: 0 out of 32 (0%)
Timing Summary:
Speed Grade: -4
Min. period: 15.675ns (Max. Frequency: 63.795MHz)
Min. input arrival time before clock: 2.571ns
Max. output required time after clock: 7.447ns
Maximum combinational path delay: No path found
```

5.5.2 Aplicação *Slave*

Seguidamente são apresentados os principais recursos utilizados da FPGA na implementação do módulo CAN1588Light(Slave), nomeadamente:

```
Final Synthesis Report
=================================================
Selected Device :xc3s1500l-4fg320
Device utilization summary:
N? of Slices: 1949 out of 13312 (14%)
N? of Slice Flip-Flops: 1774 out of 26624 (6%)
N? of 4 input LUTs: 2740 out of 26624 (10%)
N? of Bonded IOBs: 27 out of 221 (12%)
N? of Global CLKs: 2 out of 8 (25%)
N? of Block RAMs: 0 out of 32 (0%)
Timing Summary:
Speed Grade: -4
Min. period: 19.988ns (Max. Frequency: 50.031MHz)
Min. input arrival time before clock: 9.945ns
Max. output required time after clock: 11.348ns
Maximum combinational path delay: No path found
```

5.6 Utilização do módulo do ponto de vista da aplicação

Os módulos realizados permitem a fácil integração com outros projectos, dado que ocupam relativamente poucos recursos da FPGA e pelo facto de todo o processo de sincronização e manutenção de relógio ser transparente para as aplicações. Estas utilizam o CAN1588Light para terem acesso um relógio sincronizado e também para poderem enviar e receber mensagens CAN. Um exemplo disso mesmo está ilustrado na Figura 5.7.

Além do CAN1588Light fornecer os sinais de relógio e para aceder ao controlador CAN, tem também outros sinais definidos em *top level* que permitem serem ajustados pelo utilizador, nomeadamente:

No *Slave*

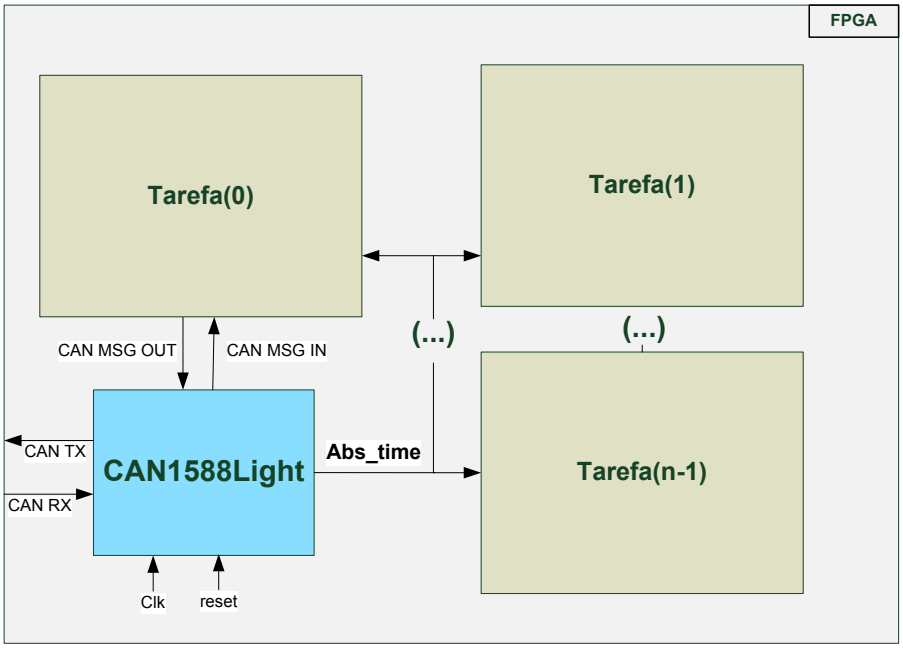


Figura 5.7: Diagrama ilustrativo da interligação do CAN1588Light com outras tarefas.

1. *deltaT*
   Número de segundos entre sincronizações.
2. *SlaveID*
   Identificador das mensagens de sincronização do *Slave*.
3. *stdID*
   Padrão dos bits mais significativos do ID para as mensagens de sincronização.
4. *OffCmp*
   Valor de *offset* abaixo do qual é feita a correcção de relógio pelo ajuste da taxa de contagem do relógio (PID).
5. KP, KI e KD
   Constantes do compensador PID.

No *Master*

1. *BlockedSeg*
   Número de segundos ao fim dos quais se o estado de sincronização não se alterar, leva a abortar esse processo de sincronização.
2. *stdID*
   Padrão dos bits mais significativos do ID para as mensagens de sincronização.

Pode-se ainda configurar os sinais do controlador CLAN, nomeadamente os sinais *SJW*, *BRP*, *TSEG1* e *TSEG2* que permitem seleccionar a taxa de transferência no barramento CAN e também permitem escolher a frequência interna de funcionamento do core CLAN[OAF05].

Capítulo 6

Resultados

6.1 Ferramentas utilizadas para as medições

No decurso da execução desta tese, foram também realizados outros pequenos trabalhos. Foi realizado um simulador do protocolo IEEE 1588 em Matlab que auxiliou na previsão e percepção do comportamento do protocolo. Também foram feitas ferramentas que permitiram obter resultados das implementações realizadas do protocolo de sincronização, nomeadamente o medidor de *offset* e o medidor de tempos de bloqueio.

Estas ferramentas são sucintamente apresentadas em anexo nas secções A.7 e A.8.

6.2 Resultados obtidos na implementação em software

Nesta secção, serão mostrados os resultados relativos à implementação do protocolo 1588Light em software, cujos os nodos do sistema distribuído são placas DETPIC. O microcontrolador usado é o PIC18F258 da Microchip a uma frequência de 20 MHz. Primeiramente serão apresentados os resultados relativos aos erros obtidos nos cristais usados e os tempos de bloqueios impostos nos nodos pelo protocolo de sincronização e manutenção do relógio. Seguidamente serão mostrados os resultados de *offset* conseguidos entre um relógio *Master* e um relógio *Slave*, prosseguindo com outras experiências que mostram o comportamento do sistema submetido a diferentes situações.

6.2.1 Erro do cristal

Durante a implementação do protocolo de sincronização em software, verificou-se que mediante o nodo que se escolhesse como *Slave* o valor de início de contagem do *timer0* (*Start_timer0*), quando o relógio estava sincronizado, era diferente. Assim como se pode ver na Tabela 6.1, quando o relógio se encontra num estado estável (sincronizado), o seu valor de *Start_timer0* oscila entre dois valores. Para o valor inferior o relógio conta abaixo da taxa desejada e para o valor superior conta acima da taxa desejada.

Partindo da diferença entre o valor ideal do *Master* e do *Slave*, é possível calcular o erro em partes por milhão (ppm) dos osciladores relativamente ao seu *Master*.

$\frac{536-518}{65000} \times 1000000 = 277ppm$

Star_Timer0	
Master	***Slave***
536	518/519
536	539/540
536	557/558

Tabela 6.1: *Start_timer0* do *Master* e de vários *Slaves* quando se encontram sincronizados.

$\frac{540-536}{65000} \times 1000000 = 62ppm$

$\frac{558-536}{65000} \times 1000000 = 338ppm$

O pior caso observado ao nível do relógio do *timer* foi um erro de 338ppm, correspondendo a 338μs/s.

6.2.2 Tempos de execução

Os tempos de bloqueio são fonte significativa da degradação do rigor de relógio numa implementação baseada em software. Os resultados obtidos poderão justificar os valores de *offset* expectáveis em situações de pior caso. Por outro lado a medida de tempo de execução permite determinar a carga de CPU imposta pelo protocolo de sincronização e pela manutenção do relógio.

Na Figura 6.1 é ilustrado o esquema de montagem relativo ao medidor de tempos de execução.

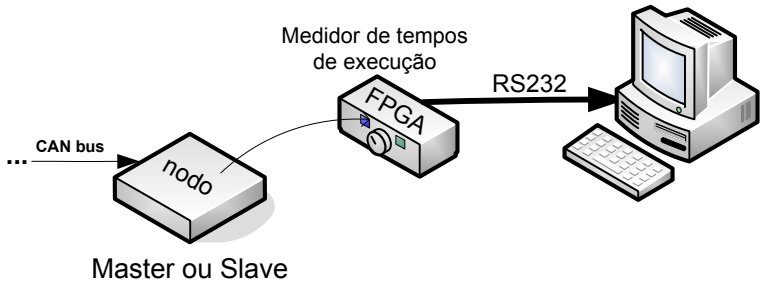


Figura 6.1: Montagem efectuada para obter os tempos de execução.

6.2.2.1 *Master*

A sincronizar 1 *Slave*

O teste foi realizado tal como mostra na Figura 6.2, ou seja, utilizando somente um *Slave* a sincronizar com o seu *Master*, tendo-se obtido os seguintes tempos de execução presentes na Figura 6.3.

Os picos presentes na Figura 6.3 representam o instante em que é realizado o processo de sincronização. Sendo que este se realiza de 2 em 2 segundos, sabe-se que, por exemplo,

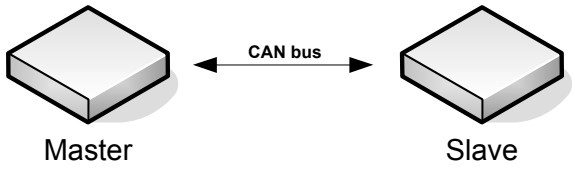


Figura 6.2: Configuração da rede utilizada para medir os tempos de execução no relógio *Master*.

Figura 6.3: Tempo de execução no *Master* quando sincroniza 1 *Slave*.

entre 11 picos decorrem $2 \times 10 = 20$ segundos.

Somando agora todos os tempos de execução nesse intervalo obtém-se 270400μs, que representam 13520μs ($\frac{270400}{20}$) por segundo.

$\frac{13520 \times 10^{-6}}{1} \times 100 = 1.35\% \implies$ Percentagem de CPU gasta no *Master* a sincronizar 1 *Slave*.

A sincronizar 3 *Slave*

Seguidamente e como mostra na Figura 6.4, vai-se medir os tempos de execução no *Master* numa rede com três *Slaves* a sincronizarem com um *Master*, tendo-se nesse caso obtido os seguintes tempos de execução presentes na Figura 6.5.

Na Figura 6.5 cada 3 picos (mais próximos) representam os instantes de sincronização do *Master* com os 3 *Slaves*. Somando todos os tempos de execução num intervalo de 20 segundos obtém-se 303360μs, que representam 15168μs ($\frac{303360}{20}$) por segundo.

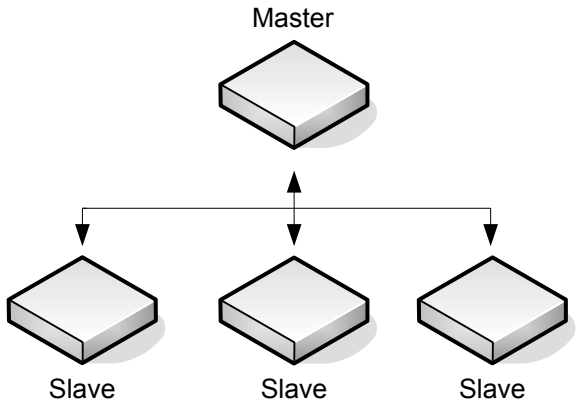


Figura 6.4: Configuração da rede utilizada para medir os tempos de execução no *Master*.

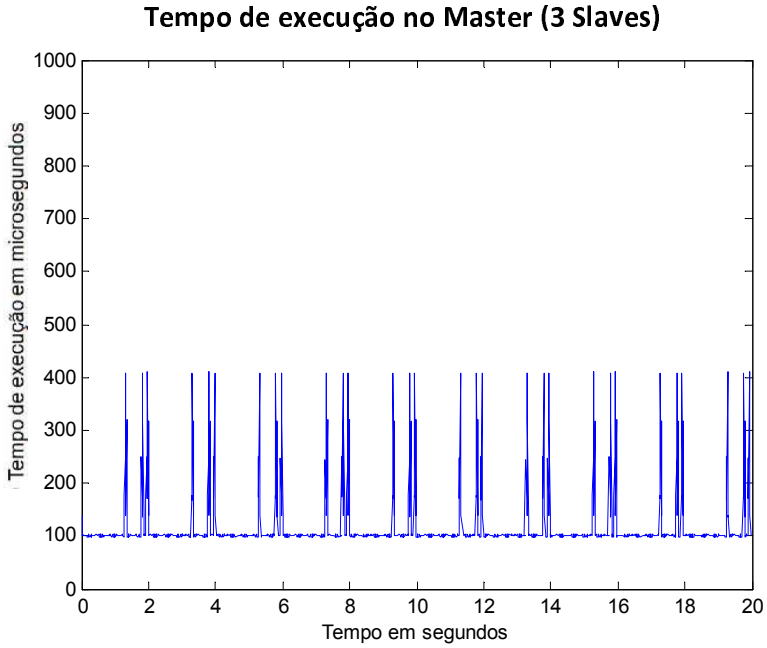


Figura 6.5: Tempo de execução no *Master* quando sincroniza 3 *Slaves*.

$\frac{15168\times10^{-6}}{1} \times 100 = 1.5\% \Longrightarrow$ Percentagem de CPU gasta no *Master* a sincronizar 3 *Slaves*.

Nota: o tempo de bloqueio ao nível da aplicação para a situação de pior caso corresponde à soma dos três picos de carga. Isto acontece quando as sincronizações dos *Slaves* ocorrem nos mesmos instantes. Neste gráfico é possível verificar que sendo os *Slaves* a iniciar o processo de sincronização essa sobreposição tem pouca probabilidade de suceder. No entanto, se fosse o *Master* a desencadear o processo de sincronização, iria ocorrer a sincronização dos vários *Slaves* com o *Master* no mesmo instante, poten-

ciando provavelmente a ocorrência de grandes picos de carga de CPU no *Master* (neste caso cerca de três vezes superiores).

6.2.2.2 *Slave*

Figura 6.6: Tempo de execução no *Slave*.

Na figura 6.6 são apresentados os tempo de execução num relógio Slave. Fazendo um raciocínio idêntico ao que foi feito para o *Master*, se forem somados todos os tempos de execução contidos entre 11 picos consecutivos, obtém-se 94258μs, que representam 4713μs por segundo.

$\frac{4713\times10^{-6}}{1} \times 100 = 0.47\% \Longrightarrow$ Percentagem de CPU gasta no *Slave*.

Este valor mantém-se constante independentemente dos número de relógios *Slaves* presentes na rede.

Na Tabela 6.2 são mostradas algumas características dos tempos de execução obtidos anteriormente.

6.2.3 Erro ao nível da aplicação

Passando agora para os resultados de *offset* propriamente ditos, começa-se com uma configuração de rede simples (um *Slave* e um *Master*) realizado posteriormente estas medições em diferentes condições da rede e dos nodos. O *offset* entre os relógios foi capturado seguindo o esquema presente na Figura 6.7.

Em todas as medidas o tempo no *Master* vai ser sempre medido ao nível do *timer*, de forma a servir de referência temporal.

A taxa de transmissão na rede CAN será de 1Mbit/s salvo em indicação contrária.

	Tempo de execução		
	***Master* 1S**	***Master* 3S**	***Slave***
Mínimo (μs)	97	97	18
Máximo (μs)	402	403	828
Desvio padrão (μs)	27.69	49.85	52.59
Média (μs)	103.99	112.34	35.79

Tabela 6.2: Tempos de bloqueio.

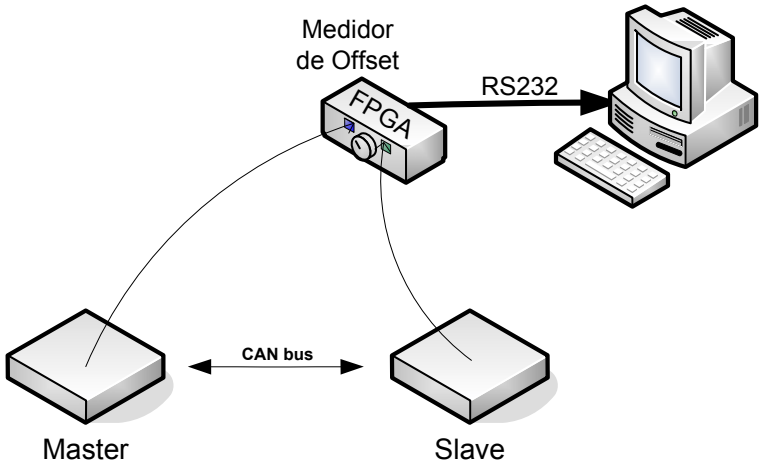


Figura 6.7: Esquema de montagem utilizado para capturar o *offset*.

6.2.3.1 *Master - Slave*

Utilizando o esquema presente na Figura 6.8 serão mostrados os resultados de *offset* com medições ao nível do *timer* e posteriormente ao nível da aplicação no nodo *Slave*.

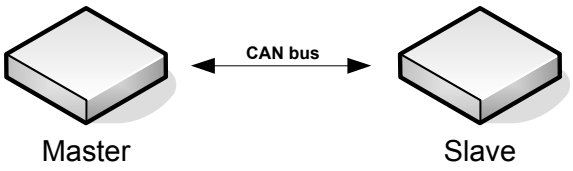


Figura 6.8: Esquema utilizado na medição de offset entre um relógio *Master* e um relógio *Slave*.

Offset ao nível do *timer*

Medindo o *offset* ao nivel do timer foi possivel obter os resultados presentes na Figura 6.9 com as caracteristicas referidas na Tabela 6.3.

É possível verificar que o *offset* varia entre valores positivos e negativos. Os flancos ascendentes de *offset* ocorrem quando o *Start_timer0* está com valor superior ao ideal

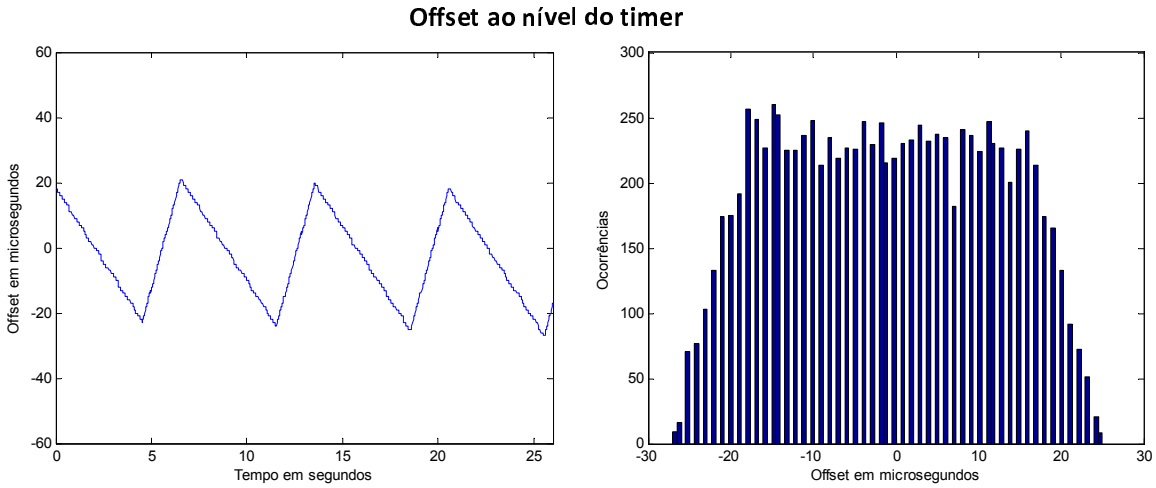


Figura 6.9: *Offset* entre um relógio *Master* e *Slave* ao nível dos seus *timers*.

Mínimo (μs)	-19
Máximo (μs)	27
Desvio padrão (μs)	11.52
Média (μs)	4.18

Tabela 6.3: Características do *offset* ao nível do *timer*.

e o flanco descendente quando o *Start_timer0* está com um valor inferior ao ideal. As mudanças de flanco representam os instante em que o compensador PID altera o valor do *Start_timer0*.

Offset ao nível da aplicação

Função *get_time*

A implementação efectuada exporta para o nível de aplicação a função *get_time*, a qual permite obter o tempo de sistema. Tal como referido na secção 4.6.1.1 a obtenção do tempo requer algumas operações aritméticas envolvendo os registos do *timer0* em hardware e a variável de software.

A função *get_time* permite ao nível da aplicação saber o tempo actual do relógio. No entanto, e tal como mostra na Figura 6.10, o tempo só é obtido em média passado 85 a 90 μs, visto que a função demora esse tempo a executar. O facto de o *timestamp* ao tempo do relógio ser feito logo no início da função é vantajoso quando necessitamos de capturar o tempo em que um evento aconteceu. Mas por outro lado, quando esta num *loop* à espera de um certo tempo, este vai aparecer sempre com um atraso, que no melhor dos casos é 85 a 90 μs depois.

O valor devolvido pela função *get_time* refere-se ao inicio da sua execução, o que pode levar a atrasos na ordem do seu tempo de execução. Por exemplo, na situação ilustrada na Figura 6.11 a aplicação pretende aguardar pelo instante t=40μs. Se a primeira invocação da função *get_time* for efectuada um pouco antes (pior caso) a função devolve

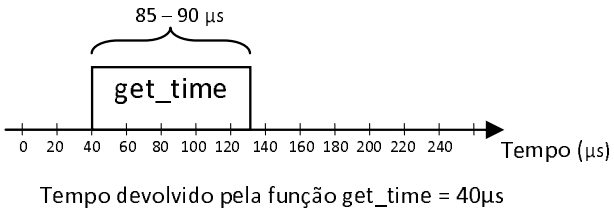


Figura 6.10: Função *get_time*.

um valor temporal inferior aos 40 μs e assim a condição fronteira ainda se apresenta como falsa (39≥40 P.F.), sendo necessário invocar novamente a função *get_time* para a seguir devolver o tempo que cumpra a condição (140≥40 P.V.). Esta situação de pior caso, faz com que exista um atraso na validação da condição de quase $2\times(90 + 9) = 189\mu$s, em que os 9μs representam o atraso gasto na validação do tempo devolvido pela função *get_time*.

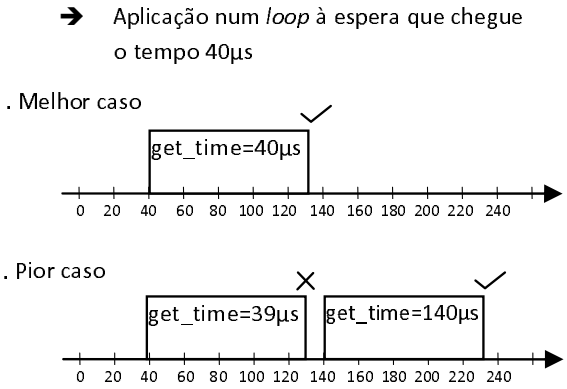


Figura 6.11: Situação de melhor e pior caso quando utilizamos a função *get_time* num *loop* à espera de um determinado tempo.

Medindo agora o *offset* ao nível da aplicação no nodo *Slave* foi possível obter o *offset* presente na Figura 6.12 com as características da Tabela 6.4. O tempo no *Slave* é conseguido recorrendo à função *get_time*.

Mínimo (μs)	-161
Máximo (μs)	27
Desvio padrão (μs)	34.43
Média (μs)	-88.43

Tabela 6.4: Características do *offset* ao nível da aplicação.

O *offset* nestas circunstâncias segue o mesmo comportamento obtido no *offset* ao nível do *timer0* sendo que revela um atraso médio de ordem do tempo de execução da função *get_time* bem como uma incerteza devido às situações entre melhor e pior na obtenção

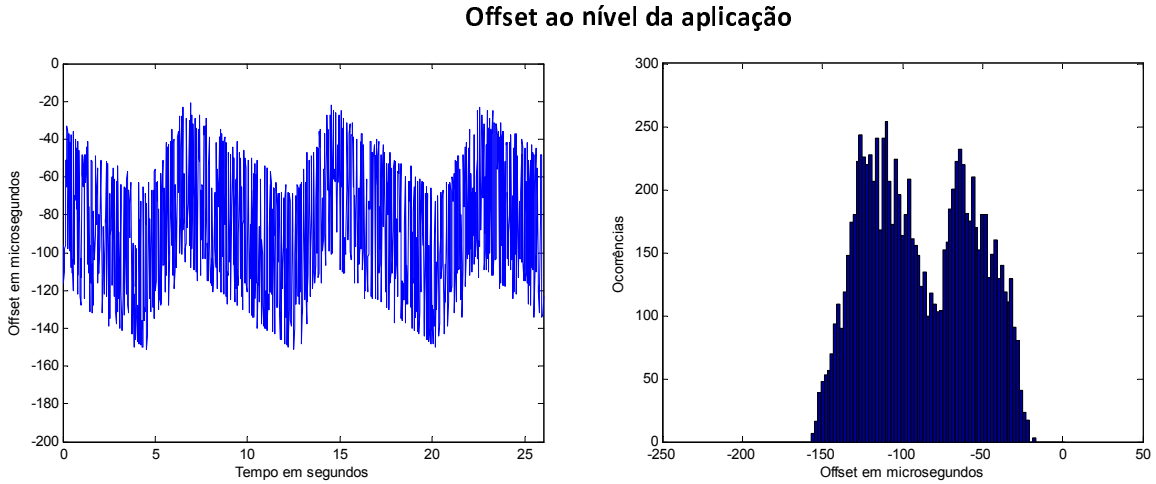


Figura 6.12: *Offset* entre um relógio *Master* e *Slave* ao nível da aplicação.

do tempo pela função *get_time* já anteriormente referidos.

Não é possível verificar um *offset* da ordem do máximo tempo de bloqueio já que este irá ocorrer sempre entre leitura de offset, não se prevendo a sua ocorrência em simultâneo.

6.2.3.2 *Master* - vários *Slaves*

O esquema de montagem utilizado para este teste foi o representado na Figura 6.13. Os resultados obtidos em relação ao *offset* são similares aos obtidos só com um *Slave* na rede. No entanto, o *Master* estará sujeito a uma maior carga no CPU devido a ter que sincronizar com os vários *Slaves*.

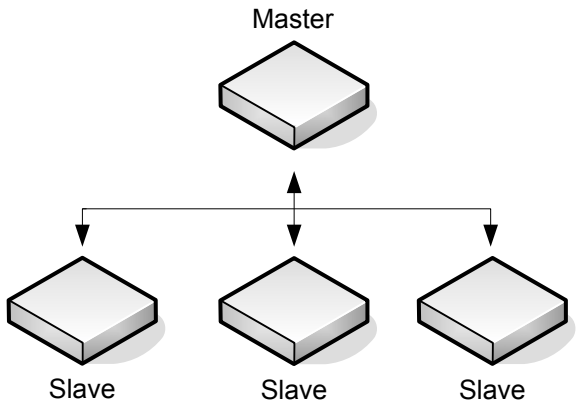


Figura 6.13: Esquema utilizado para na medição de offset entre um relógio *Master* e vários relógios *Slave*.

6.2.3.3 *Master* - *Slave* com carga no barramento

Com o objectivo de avaliar o efeito da carga no barramento na sincronização de relógio, usou-se um esquema com um *Master* e um *Slave* na rede, ao qual foi acrescentado um nodo que apenas serve para adicionar tráfego na rede, tal como pode ser visto na Figura 6.14. As mensagens enviadas pelo *Traffic generator* têm um ID inferior (mais prioritária) ao das mensagens do 1588Light, de forma a verificar o impacto destas no rigor do relógio.

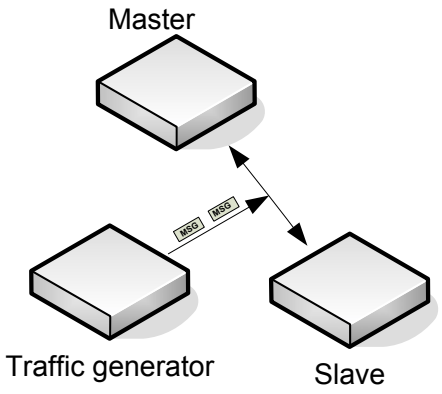


Figura 6.14: Esquema utilizado na medição de *offset* entre um relógio *Master* e um relógios *Slave*, mediante diferentes condições de carga de ocupação do barramento.

As cargas impostas e o respectivo período de envio das mensagens está presente na Tabela 6.5. Cada mensagem ocupa 118 μs a 1Mbit/s (já com *stuff bits* e o IFS - *Inter Frame Space*), sendo que esta medida foi obtida utilizando um osciloscópio digital. Também foi testado o *offset* para uma percentagem de barramento superior a 100%.

Percentagem de barramento ocupada	**Período de envio de mensagens**
20	575μs
40	287μs
60	191μs
80	143μs
≈100	118μs

Tabela 6.5: Período de envio de mensagens no barramento mediante carga percentual ocupada.

O padrão de offset manteve-se constante com a introdução de tráfego no barramento não se verificando alterações significativas nas características de offset presentes na Tabela 6.6.

Com uma ocupação de barramento muito próxima dos 100%, o tempo de execução do protocolo aumenta dos típicos 25 ms, para 200 a 400 ms.

Se a ocupação do barramento com mensagens de ID mais prioritário for superior a 100%, o relógio não consegue sincronizar.

	Tráfego no barramento				
	20%	**40%**	**60%**	**80%**	**≈100%**
Mínimo (μs)	-152	-156	-157	-156	-162
Máximo (μs)	-20	-17	-22	-22	-18
Desvio padrão (μs)	33.69	33.25	33.17	33.69	34.25
Média (μs)	-86.94	-87.29	-90.50	-88.93	-90.54

Tabela 6.6: Características do *offset* mediante diferentes cargas no barramento.

6.2.3.4 *Master - Slave* com *kernel* RTKPIC

6.2.3.5 Tempo de execução do *kernel*

A utilização de um *kernel* a decorrer num nível de interrupção mais prioritário que o protocolo de sincronização irá necessáriamente ter impacto na sincronização de relógio. Este provocará dos maiores atrasos na obtenção do tempo ao nível da aplicação bem como na obtenção dos *timestamps* relativos ao protocolo de sincronização. Na Figura 6.15 é possível observar os diferentes tempos de execução mediante o número de tarefas que *kernel* tem de escalonar.

Na Tabela 6.7 são apresentadas algumas características dos tempos de execução anteriormente mostrados.

Tarefas	**1**	**3**	**6**
Mínimo (μs)	31	31	31
Máximo (μs)	417	501	627
Desvio padrão (μs)	51.22	62.22	78.15
Média (μs)	38.78	40.23	42.36

Tabela 6.7: Tempos de bloqueio impostos pela utilização do *kernel*.

Na Figura 6.16 e na Tabela 6.8 são mostradas os resultados de offset com as respectivas características mediante um diferente número de tarefas a escalonar pelo *kernel*.

	Tarefas		
	1	**3**	**6**
Mínimo (μs)	-380	-487	-635
Máximo (μs)	16	44	37
Desvio padrão (μs)	37.69	41.25	49.13
Média (μs)	-81.94	-89.27	-92.47

Tabela 6.8: Características do *offset* mediante o número de tarefas a executar pelo *kernel*.

Com estes resultados verifica-se que a utilização de um *kernel* num nível de prioridade superior degrada o *offset* entre os relógios devido aos atrasos acrescidos que podem sofrer os *timestamps* levando a uma ligeira deformação do sinal de *offset* em relação a resultados

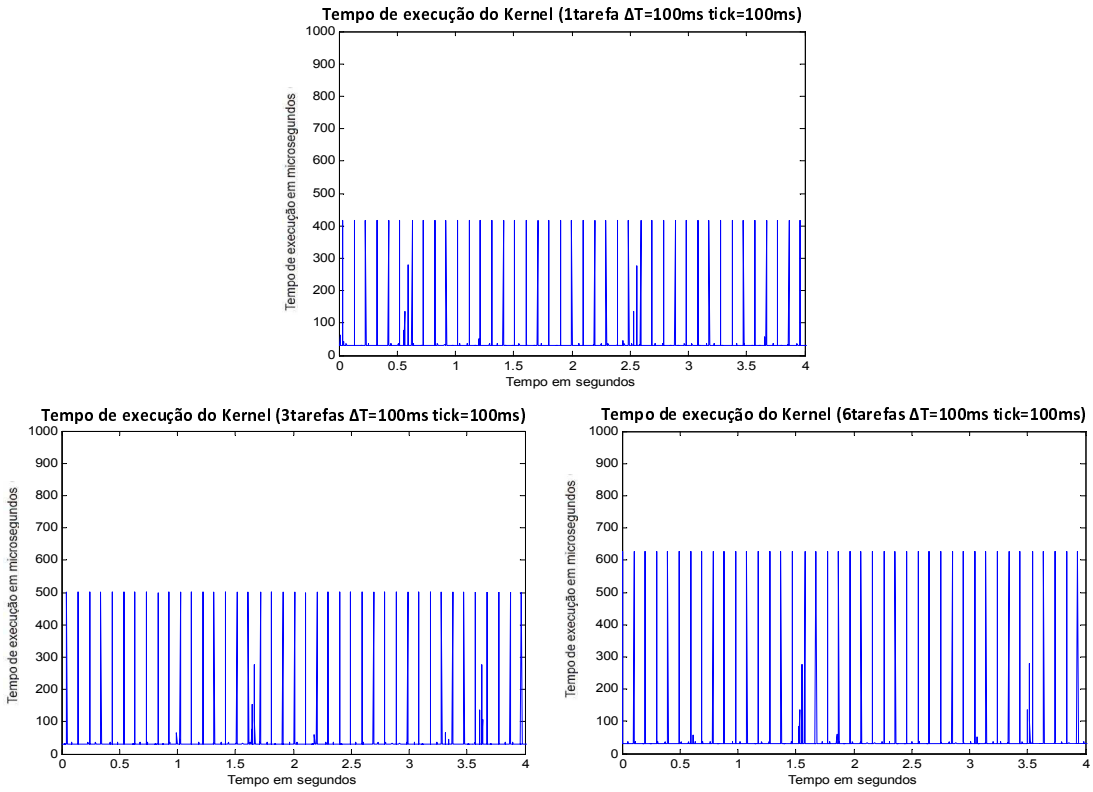


Figura 6.15: Gráficos dos tempos de execução do *kernel* no nodo *Slave*.

anteriores. Ao nível da aplicação poderão ocorrer picos de *offset* devido aos tempos de bloqueio impostos pela execução do *kernel*.

No caso em questão, conclui-se que quanto maior for o número de tarefas a escalonar pelo *kernel* pior será o sinal de offset entre os relógios bem como maiores serão os picos presentes no sinal de *offset*.

Também foi realizada uma experiência com *ticks* e tarefas de 10 em 10 ms em que se verificou (comparando com estes resultados) uma maior degradação de sinal bem como uma maior probabilidade do aparecimento de picos de *offset*.

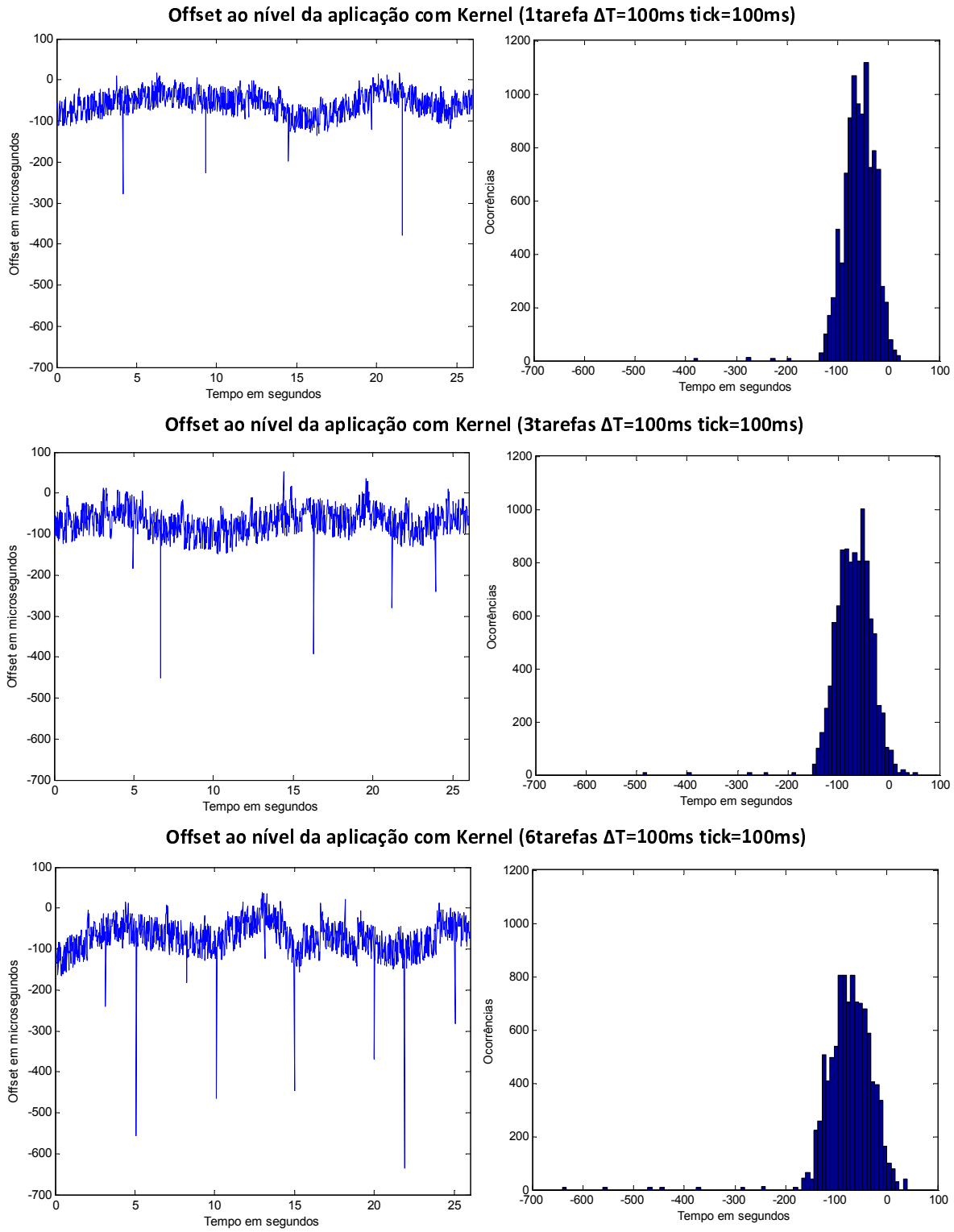


Figura 6.16: Gráficos dos tempos de bloqueio impostos pelo *kernel* no nodo *Slave*.

6.3 Resultados obtidos na implementação de hardware

Nesta secção, serão mostrados os resultados relativos à implementação do protocolo 1588Light em hardware, cujos os nodos do sistema distribuído são duas placas RC10 da Celoxica. A FPGA contida nestas placas é uma Xilinx Spartan 3L XC3S1500L-4-FG320.

6.3.1 Fase de inicialização

Após o ajuste inicial do relógio este tende a manter-se sincronizado utilizando para o ajuste do sistema de relógio um compensador PID. Este provocará numa fase de inicialização o comportamento presente na Figura 6.17.

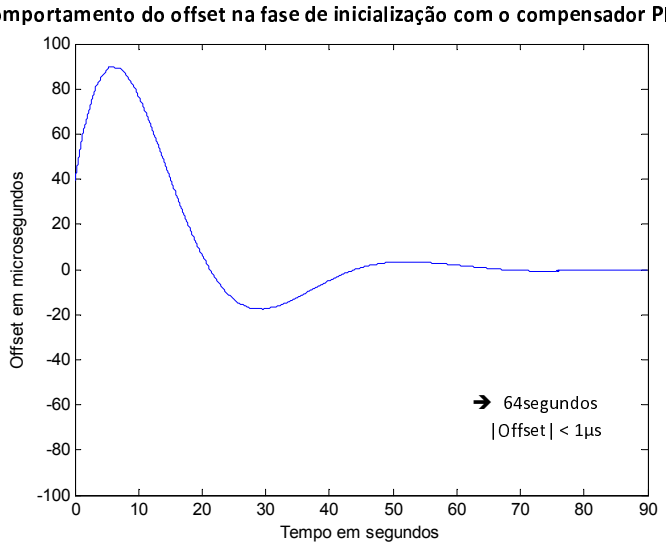


Figura 6.17: *Offset* entre um *Master* e um *Slave* obtido na fase de inicialização com o compensador PID.

O tempo que o *offset* demora a ficar no estado estacionário depende das constantes (KP, KI e KD) utilizadas no compensador PID. Com valores altos nas constantes o *offset* tende mais rapidamente para o valor zero (regime transitório mais curto ≈30 segundos) mas tornará o sistema relativamente menos estável a variações do sistema. No entanto com valores mais pequenos para as constantes, o offset demora mais tempo tender para zero (regime transitório mais longo ≈80 segundos) mas é consideravelmente mais estável permitindo também alterações mais suaves na correcção do offset no relógio.

Este comportamento também foi observado na implementação em software sendo que apenas diferiu na gama de valores offset obtidos (centenas de microsegundos).

6.3.2 Erro do cristal

Ao ajustar o relógio sem utilizar o compensador PID, ou seja, corrigindo apenas o valor o relógio não alterando a frequência de funcionamento do mesmo foi possível verificar que passados 2 segundos (instantes de sincronização) o offset era de 38μ. Isto leva a concluir que o desvio do relógio é de 19μs/s.

6.3.3 Offset

6.3.3.1 Sem a utilização do compensador

Não utilizando o compensador PID e apenas ajustando o valor do relógio tal como vem referido na norma, foi obtidos os resultados de *offset* presentes na Figura 6.18, cujas as características são presentadas na Tabela 6.9.

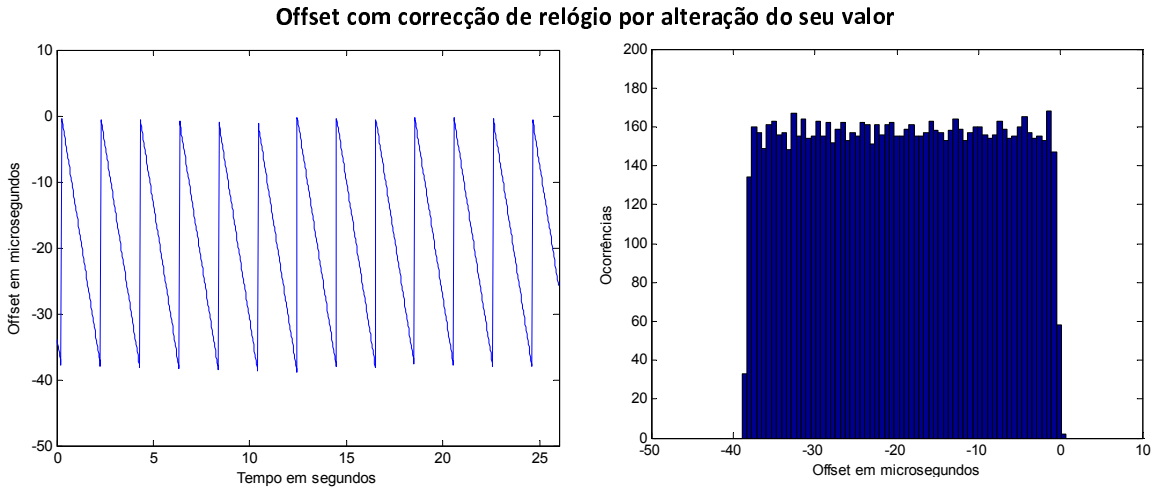


Figura 6.18: *Offset* entre um *Master* e um *Slave* ajustando apenas o valor do relógio.

Mínimo (μs)	-38.82
Máximo (μs)	0.3
Desvio padrão (μs)	10.98
Média (μs)	-19.30

Tabela 6.9: Características do *offset* com o ajuste pelo valor do relógio.

Através da análise do gráfico anterior é possível ver que mesmo o relógio *Slave* corrigindo o valor do seu relógio periodicamente este irá sempre desviar-se pois tem um taxa de contagem ligeiramente diferente do seu *Master*, fazendo com que *Slave* de 2 em 2 segundos em média esteja desfasado de 38μs.

6.3.3.2 Com a utilização do compensador

Agora são apresentados os resultados utilizando o compensador PID por forma a ajustar a taxa de contagem do relógio no bloco *Cnt_utick*. Na Figura 6.19 e 6.20 são mostrados os resultados de *offset* para valores altos e para valores baixos das constantes do compensador PID, respectivamente. Na tabela 6.10 são apresentadas algumas características destes resultados de *offset*.

Destes resultados pode-se concluir os resultados utilizando o compensador PID são significativamente melhores e mais estáveis do que na situação em que apenas se ajusta o valor de relógio. Comparando agora os resultados relativos à utilização de diferentes constantes no

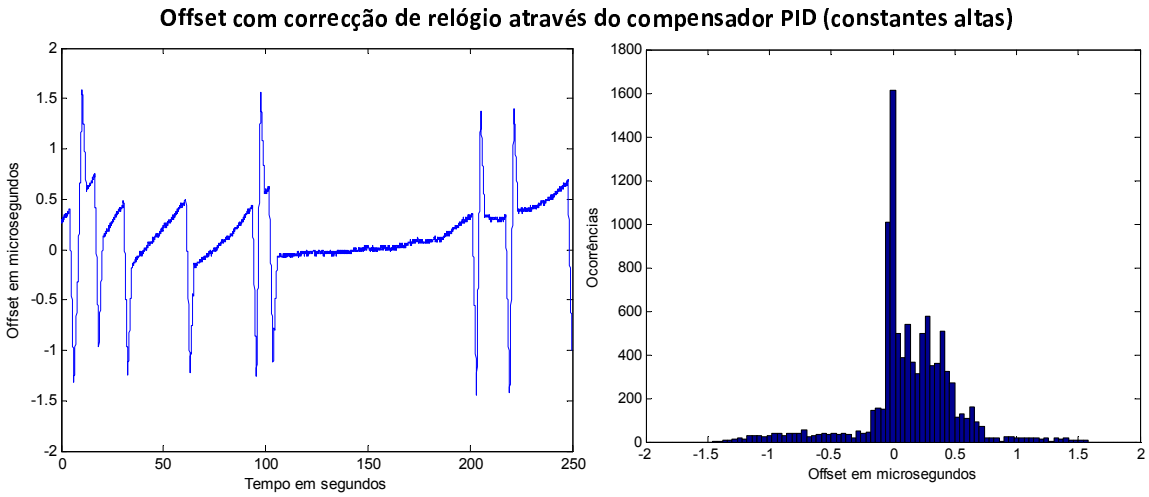


Figura 6.19: *Offset* entre um *Master* e um *Slave* obtido com constantes do PID altas.

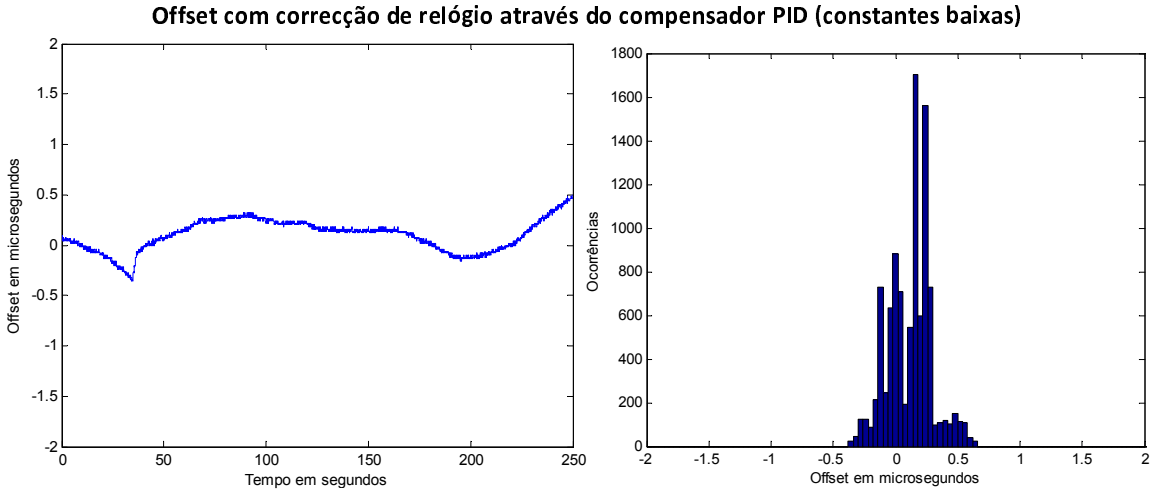


Figura 6.20: *Offset* entre um *Master* e um *Slave* obtido com constantes do PID baixas.

	Valor das constantes PID	
	altas	**baixas**
Mínimo (μs)	-1.44	-0.36
Máximo (μs)	1.58	0.64
Desvio padrão (μs)	0.383	0.176
Média (μs)	0.123	0.119

Tabela 6.10: Características do *offset* mediantes diferentes valores para as constantes do compensador PID.

compensador verifica-se que para valores mais baixos o *offset* apresenta melhores resultados e também mais estáveis. No entanto, tal como já referido anteriormente, esta maior estabilidade paga-se com um maior período transitório, ou seja, o *offset* demorará mais tempo até atingir um valor estável.

O comportamento de offset ao longo do tempo aparece relativamente irregular dado que o relógio tem apenas resolução de microsegundos não se conseguindo por isso que o compensador PID ajuste o valor de *offset* correctamente para valores inferiores ao microsegundo.

6.3.3.3 Análise funcional

Executaram-se também testes semelhantes ao realizados para a implementação em software.

Num primeiro teste colocaram-se 3 tarefas a correr nos nodos, não se observando nenhum impacto tal como se esperava. Isto acontece devido à possibilidade de executar múltiplas tarefas em simultâneo numa FPGA, não existindo por isso a problemática dos tempos de bloqueio das tarefas na degradação do rigor do relógio.

Posteriormente, foi introduzido tráfego no barramento CAN, não se verificando alterações significativas nos resultados de *offset*. Isto verifica-se sobretudo quando as constantes do compensador PID são valores baixos, pelo que possíveis atrasos na execução do protocolo não provocarão variações significativas na taxa de contagem do relógio.

6.4 Análise comparativa

Na implementação do protocolo em software os resultados ao nível do *timer* são bastante melhores do que os resultados ao nível da aplicação. Ao nível do *timer* consegue-se um *offset* dentro das poucas dezenas (+/- 30) de microsegundos. Na aplicação devido aos tempos de bloqueios impostos pelo protocolo de sincronização de relógio e manutenção do mesmo na rotina de interrupção, os resultados tipicamente rondam as poucas centenas (-200 a 0) de microsegundos.

Além do *offset* ser significativamente superior em relação à medição no timer, este vem sempre atrasado no melhor dos casos 85-90 μs devido ao tempo que a função *get_time* demora a executar. Isto explica porque o valor médio dos valores de *offset* se situa aproximadamente entre 85 e 90 μs.

Na implementação em hardware o acesso ao relógio é feito em hardware, não sofrendo por isso atrasos tal como acontece na implementação em software.

Relativamente ao comportamento do protocolo com diferentes cargas de ocupação no barramento, verificou-se que este se apresenta bastante estável mesmo com taxas de ocupação bastante elevadas, sendo que apenas se notou um aumento perceptível da duração de execução do protocolo quando a carga de ocupação estava muito próxima dos 100%. Neste contexto a implementação em software e hardware equiparam-se.

No entanto dado que o rigor conseguido na implementação em hardware é das poucas unidades de microsegundo, dependendo do valor das constantes utilizadas no compensador PID é possível observar uma ligeira degradação do *offset* devido às variações da duração do protocolo. Estas iriam fazer variar o intervalo entre actuações no relógio, afectando assim o rigor do mesmo. No entanto se o compensador tiver constantes baixas o sistema torna-se lento e mais estável não ficando tão sensível às variações de duração do protocolo.

No que toca à afectação do rigor do relógio mediante a realização de outras tarefas nos nodos, os resultados são bem distintos. Na FPGA (implementação em hardware) não tem qualquer impacto, pois esta tem a capacidade de realizar múltiplas tarefas em simultâneo, não existindo por isso a problemática dos tempos de bloqueio das tarefas na degradação do rigor do relógio.

Já na implementação em software do protocolo de sincronização, esta é directamente afectada por tarefas que tenham mais prioridade que o protocolo, em que na implementação do PIC, é representada pela rotina de interrupções de alta prioridade (utilizada pelo *Kernel*). O seu impacto no rigor do relógio será tanto maior quanto maiores e com maior frequência forem os tempos de bloqueios impostos por essas aplicações. Todavia, se as tarefas forem menos prioritárias que o protocolo, estas não afectarão o rigor do relógio.

Em suma, os resultados da implementação do protocolo de sincronização em hardware foram melhores e mais estáveis, mesmo quando sujeitos a condições adversas.

Capítulo 7

Conclusões

Este capítulo conclui a dissertação, começando por apresentar um resumo do trabalho realizado no âmbito desta tese, sendo de seguida feita uma análise final dos resultados obtidos.

Finalmente, são referidos alguns pontos de trabalho futuro, nomeadamente na optimização e incremento de funcionalidades aos sistemas desenvolvidos.

7.1 Resumo do trabalho realizado

No âmbito desta dissertação foi proposta a implementação de um protocolo derivado do protocolo IEEE 1588 em redes CAN. Algumas características do protocolo CAN, como a largura de banda limitada e o pequeno tamanho dos pacotes aliado ao facto de na maior parte dos casos os sistemas que usam CAN serem microcontroladores ou sistemas embutidos de baixo custo e com pouca capacidade de processamento, levou à adaptação do protocolo IEEE 1588.

Esta adaptação resultou numa simplificação feita ao nível dos campos necessários enviar por cada mensagem de sincronização, bem como na eliminação do algoritmo de selecção automática do *Master* na rede, considerando-se que o *Master* da rede é um nodo predefinido.

A implementação do protocolo em redes CAN, foi efectuada em software e em hardware.

Primeiramente, o protocolo foi implementado ao nível de software. A plataforma de desenvolvimento utilizada é baseada nas placas DETPIC, integrando um controlador PIC18F258, com controlador CAN incorporado. A partir das bibliotecas disponibilizadas, em linguagem "C"para acesso ao controlador CAN, foi implementado o protocolo ao nível da interrupção.

Esta implementação foi desenvolvida de forma a poder ser utilizada em outras aplicações. Para permitir esta utilização de forma modular desenvolveu-se um módulo de interrupções que permite a integração de várias funções dentro da mesma ISR, sem no entanto ser necessário o acesso ao seu código.

Posteriormente, depois de tirados os resultados da implementação em software, foi realizada a implementação do protocolo em hardware. Nesta foi utilizado o controlador CAN (CLAN), disponibilizado na forma de núcleo de propriedade intelectual modelado em VHDL, sintetizável e implementável em FPGAs da Xilinx. A placa de desenvolvimento utilizada foi a RC10 da Celoxica com uma FPGA Spartan-3 e toda a implementação foi desenvolvida em linguagem VHDL. Foram realizados módulos de sincronização (*Master* e *Slave*) independentes, parametrizáveis e sintetizáveis. Estes módulos associados ao multiplexador de mensagens desenvolvido e ao CLAN, tornam o sistema totalmente transparente para o utilizador fornecendo o acesso a um relógio sincronizado bem como o acesso ao controlador CAN.

7.2 Análise dos resultados

Na adaptação do protocolo IEEE 1558 para o 1588Light, resultou um algoritmo de sincronização leve e com necessidade de troca de poucas mensagens CAN, sendo por isso ideal para aplicar microcontroladores ou sistemas embutidos de baixo custo. Prova disso é o facto de a carga de CPU gasta pelo protocolo de sincronização ser menor do que 0.5% num PIC a 20Mhz, e carga ocupado no barramento ser de 0.022% @ 1Mbit/s (com apenas um *Slave* na rede).

Outro aspecto importante, sempre tido em conta, foi tornar estes módulos de sincronização passíveis de serem facilmente utilizados em outros projectos. Esta portabilidade na implementação do 1588Light em software resulta de se ter desenvolvido um módulo de interrupções (secção A.6) que permite a integração de outras funções na rotina de interrupção. Outro aspecto tido em atenção reside no facto de se terem utilizado constante globais. Alterando estas, o módulo de sincronização pode ser facilmente portado para CPUs a funcionar a uma frequência diferente, bem como utilizado em barramentos com uma taxa de transmissão diferente da utilizada.

Na implementação em hardware a portabilidade é mais evidente, dado que a implementação resultou em módulos independentes, sintetizáveis, parametrizáveis e implementáveis em FPGAs da Xilinx. Os módulos de sincronização CAN1588Light (*Master* e *Slave*) podem ser usados de forma transparente pelo utilizador, sem este necessitar qualquer tipo de intervenção na execução do protocolo ou na gestão para o acesso ao controlador CAN.

Ao nível da implementação em software conseguiram-se precisões na gama das poucas dezenas de microsegundos ao nível do *timer* dos microcontroladores. Ao nível da aplicação os resultados de *offset* obtidos estavam na ordem das poucas centenas (-200 a 0) de microsegundos.

Além do *offset* ser significativamente superior em relação à medição no *timer*, este vem sempre atrasado no melhor dos casos 85-90 μs devido ao tempo que a função *get_time* demora a executar. Isto explica porque o valor médio dos valores de *offset* se situa aproximadamente entre 85 e 90 μs.

Verificou-se um bom comportamento do protocolo quando sujeito a diferentes cargas no barramento bem como pela adição de nodos *Slaves* na rede. Quando introduzido o *kernel*, os tempos de bloqueio impostos por este, degradaram significativamente o relógio passando o offset a ser de algumas centenas de microsegundos, dependendo esta degradação do número de tarefas geridas pelo *Kernel* bem como pela frequência com que estas ocorrem. Todavia, se as tarefas forem menos prioritárias (a correr em *background*) que o protocolo não afectarão o rigor do relógio.

Na implementação em hardware os resultados obtidos estão na gama das unidades (+/- 2) de microsegundos. A gama de offset manteve-se constante com a introdução de carga no barramento, bem como com a adição de outros nodos *Slave* na rede. Nas FPGAs a execução de outras tarefas juntamente com o protocolo de sincronização não tem qualquer impacto, devido à possibilidade de nestas executar múltiplas tarefas em simultâneo, sendo que assim o problema dos tempos de bloqueio na afectação do rigor do relógio não se levanta.

Devido ao rigor dos *timestamps* conseguidos na implementação em hardware os resultados foram significativamente melhores e mais estáveis em relação à implementação em software, mesmo quando sujeitos a condições adversas.

7.3 Trabalho futuro

Após a realização desta dissertação, alguns pontos ficam em aberto, sendo possível a optimização de alguns aspectos, bem como a integração de funcionalidades adicionais nos sistemas desenvolvidos.

Ao nível da tolerância de falhas, este sistema está limitado pela falha do nodo *Master*. Assim seria desejável introduzir um mecanismo que tivesse em conta a falha do *Master*. Existem duas possibilidades pensadas. A primeira abordagem seria acrescentar, ao algoritmo actual, um mecanismo que detectasse a falha do nodo *Master*, fazendo nesse caso desencadear um processo de arbitragem para seleccionar um novo *Master*. Este iria de encontro ao que é feito no IEEE 1588, mas no entanto iria complicar e tornar mais pesado o algoritmo nos nodos. O facto de ser mais pesado provocaria maiores tempos de bloqueio, degradando a qualidade de sinal de relógio, tendo por isso alguns entraves em seguir esta abordagem.

A outra abordagem possível seria colocar vários relógios *Master* na rede, sendo que só um deles estaria activo, enquanto que os outros ficariam à escuta no barramento até que detectasse a falha do nodo *Master* activo. Assim conseguiria-se um sistema mais robusto,

dado que para o sistema falhar seria necessário que todos os relógios *Master* presentes na rede falhassem. Esta situação é muito menos provável de acontecer do que quando só existe um *Master* na rede. Ao nível dos nodos *Master* seria necessário implementar um algoritmo que permitisse fazer uma arbitragem entre eles. O factor de decisão, poderia ser a qualidade do relógio ou através de prioridades previamente atribuídas a cada nodo *Master*. Ao nível do *Slave* esta abordagem não implicaria a alteração do algoritmo implementado.

Um mecanismo muito semelhante a este foi proposto na conferência anual do IEEE 1588 em 2006 [LGK06], sendo que neste os nodos *Master* formam um *cluster* designado por Syn1588, com ligações ponto a ponto entre os vários *Masters*.

Ao nível da implementação do protocolo de sincronização em software, no PIC18F258, poderá tentar-se melhorar o rigor entre os relógios, apesar dos resultados obtidos serem bastante satisfatórios e estarem dentro do que era esperado. Um possível melhoramento seria utilizar o timer2 para efectuar o *timestamp* às mensagens de sincronização. Isto seria uma mais valia dado que este *timer* permite guardar o valor de *timestamp* do instante em que a interrupção ocorre, conseguindo-se assim um melhor rigor na obtenção do *timestamp* e consequentemente um relógio mais exacto. Não se tirou partido deste *timer*, dado que havia interesse em utilizar o módulo de sincronização juntamente com o *kernel*, mas este tal como refere na secção 4.4 tira partido do *timer2* pelo que não foi possível a sua utilização no módulo de sincronização.

Na implementação em hardware poderá mapear-se as constantes relativas às implementações em espaço de registos de forma a facilitar a alteração do seu valor e permitir a fácil interligação com outros dispositivos. Esta tem uma importância significativa quando se pretende interligar o módulo com processadores permitindo assim que estes possam alterar os valor das constantes de forma relativamente simples.

Nas implementações realizadas poderão ainda ser efectuadas mais medições e mediante outro tipo de condições. Estes deverão efectuar-se durante mais tempo de forma a comprovar que os comportamentos obtidos se verificam por períodos bastante mais longos.

Em suma, os principais objectivos deste trabalho foram alcançados. No entanto, o trabalho realizado é passível de ser optimizado e de serem acrescentadas novas funcionalidades.

Apêndice A

Anexos

A.1 Lista de Acrónimos

ASIC: Application Specific Integrated Circuit
CAD: Computer Aided Design
CAE: Computer Assisted Engineering
ISE: Integrated Synthesis Environment
CAN: Controller Area Network
I/O: Input/Output (Entrada/Saida)
PTP: Precision Time Protocol (Protocolo de precisão temporal)
PID: Proportional-Integral-Derivative (Proporcional-Integral-Derivativo)

A.2 Significado de termos comuns utilizados no IEEE 1588

***PTP port*:** Ponto de acesso lógico para as comunicações 1588 do relógio que contém o porto.

***PTP message*:** São cinco os tipos de mensagens definidas pelo 1588: *Sync*, *Follow-up*, *Delay_Request*, *Delay_Response* e a de *Management*.

***Clock*:** Um dispositivo que fornece a medida da passagem do tempo. Existem dois tipos de relógios no IEEE 1588: *Boundary Clocks* e *Ordinary Clocks*.

***Boundary clock*:** relógio com mais do que um porto PTP, com cada porto PTP a proporcionar o acesso a diferentes caminhos de comunicação PTP. Os *Boundary clock* são utilizados para eliminar as flutuações produzidas pelos *routers* e equipamentos de rede similar.

***Ordinary Clock*:** Relógio com um único porto PTP, tendo um único caminho de comunicação.

***Grandmaster Clock*:** Dentro de um conjunto de relógios é aquele que irá ser a primeira fonte de tempo para que todos os relógios fiquem sincronizados.

***Master Clock*:** Um sistema de relógios 1588 pode ser segmentado em regiões separadas por *Boundary Clocks*. Dentro de cada região haverá apenas um relógio *Master*, servindo como primeira fonte de tempo para essa região. Estes relógios *Master* irão sincronizar por sua vez com outros relógio Mater e finalmente com o *Grandmaster Clock*.

***Synchronized Clocks*:** Dois relógios estão sincronizados a uma incerteza específica se tiverem a mesma época e medidas em qualquer intervalo de tempo, mas ambos possuem uma diferença que nunca é superior à incerteza.

***Epoch*:** Referência de tempo definida pela origem da escala do tempo.

***Direct communication*:** A informação da comunicação PTP entre dois relógios sem intervenção de nenhum *Boundary clock*.

***Clock timestamp point*:** O 1588 requer a geração de *timestamp* na transmissão e recepção de todas as mensagens Sync e Delay_Req. O ponto na *stack* protocolo onde o *timestamp* é gerado é designado por ponto de *timestamp* do relógio.

***Message timestamp point*:** As mensagens *Sync* e *Delay_Req* no 1588 tem funções distintas. O ponto de *timestamp* da mensagem serve como ponto referencia dessas mensagens. Quando uma mensagem passa pelo ponto de *timestamp* do relógio, o *timestamp* é gerado e usado pelo 1588 para calcular as correcções necessárias no relógio local.

***Multicast communication*:** O 1588 requer que as mensagens PTP comuniquem via *multicast*. Neste estilo de comunicação qualquer nodo pode enviar uma mensagem e todos os nodos do mesmo segmento do subdomínio irão receber essa mensagem. Os *Boundary clocks* definem os segmentos dentro de um subdomínio.

A.3 Outros termos

***Multi-Drop*:** Conexão de vários dispositivos num canal de comunicação único.

***Timestamp*:** Tempo observado a quando da ocorrência de um evento.

***Overhead*:** Em ciências de computação é geralmente considerado como qualquer processamento ou armazenamento em excesso, seja em tempo de computação, de memória, de largura de banda ou qualquer outro recurso que seja necessário para ser utilizado ou gasto para executar uma dada tarefa.

A.4 Campos das mensagens de sincronização da norma IEEE 1588

8.5.1 Delay_Resp message field specifications

The field specifications for Delay_Resp messages are given in the following subclauses. The fields of Delay_Resp messages shall be marshaled into its on-the-wire format in the following order:

a) versionPTP
b) versionNetwork
c) subdomain
d) messageType
e) sourceCommunicationTechnology
f) sourceUuid
g) sourcePortId
h) sequenceId
i) control
j) flags
k) reserved
l) delayReceiptTimestamp
m) requestingSourceCommunicationTechnology
n) requestingSourceUuid
o) requestingSourcePortId
p) requestingSourceSequenceId

8.3.1 Sync and Delay_Req message field specifications

The field specifications for both Sync and Delay_Req messages are identical except as noted in the following subclauses. The fields of both Sync and Delay_Req messages shall be marshaled into their on-the-wire format in the following order:

a) versionPTP
b) versionNetwork
c) subdomain
d) messageType
e) sourceCommunicationTechnology
f) sourceUuid
g) sourcePortId
h) sequenceId
i) control
j) flags
k) reserved
l) originTimestamp
m) epochNumber
n) currentUTCOffset
o) grandmasterCommunicationTechnology
p) grandmasterClockUuid
q) grandmasterPortId
r) grandmasterSequenceId
s) grandmasterClockStratum
t) grandmasterClockIdentifier
u) grandmasterClockVariance
v) grandmasterIsPreferred
w) grandmasterIsBoundaryClock
x) syncInterval
y) localClockVariance
z) localStepsRemoved
aa) localClockStratum
ab) localClockIdentifier
ac) parentCommunicationTechnology
ad) parentUuid
ae) parentPortField
af) estimatedMasterVariance
ag) estimatedMasterDrift
ah) utcReasonable

8.4.1 Follow_Up message field specifications

The field specifications for Follow_Up messages are given in the following subclauses. The fields of Follow_Up messages shall be marshaled into its on-the-wire format in the following order:

a) versionPTP
b) versionNetwork
c) subdomain
d) messageType
e) sourceCommunicationTechnology
f) sourceUuid
g) sourcePortId
h) sequenceId
i) control
j) flags
k) reserved
l) associatedSequenceId
m) preciseOriginTimestamp

A.5 Controller Area Network

A.5.1 Aparecimento CAN/Histórico

O aumento da electrónica automóvel resulta de os consumidores desejarem melhor segurança, mais conforto e também do facto de os governos exigirem o melhoramento do controlo de emissões de poluentes para o meio ambiente e redução do consumo de combustível. Para isso foi necessária a introdução de funções no sistema que resolvessem estes problemas.

Estas funções necessitavam da troca de dados sendo esta feita através de linha de sinal dedicadas. Mas por motivos tais como, o aumento da complexidade das funções, o aumento do número de sistemas a controlar e consequentemente o aumento do número de cabos necessários (que representavam custo, peso e espaço ocupado) fez com que se tornasse vantajoso adoptar outra forma de ligar os dispositivos. A solução adoptada foi a utilização de um barramento de dados série, tal como se pode perceber pela Figura A.1. Esta foi a razão pela qual a Bosch na década de 80's desenvolveu o *Controller Area Network* (CAN), que foi standardizado internacionalmente (ISO 11898) em 1994 e desde então tem sido utilizado por inúmeros construtores de semicondutores (*cast in silicon*). Isto levou a um crescimento progressivo do número de nodos CAN vendidos, tal como se pode ver no gráfico da Figura A.2.

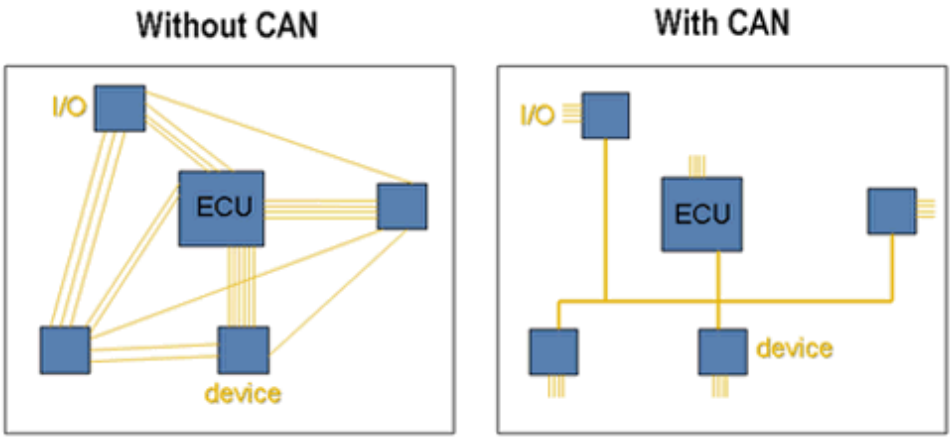


Figura A.1: Diferenças entre um sistema com ligações ponto-a-ponto e outro com ligações através de um barramento CAN.

Usando CAN é possível ligar estações (controladores, sensores e actuadores) através de um barramento série. Este protocolo que corresponde à camada física e MAC no modelo ISO/OSI, satisfaz as exigências de tempo-real das aplicações automóveis. Dado que o Protocolo CAN apenas é referente à camada física e camada MAC posteriormente surgiram outros protocolos de mais alto nível para usarem o CAN tais como o *CANopen* e *DeviceNet*. Outras marcas têm adoptado largamente estes protocolos adicionais, que hoje em dia são *standards* para as comunicações industriais e indústria automóvel.

A.5.2 Regras da arbitragem

O barramento é considerado como livre depois da transmissão da mensagem mais o espaço necessário entre mensagens. No caso de concorrerem várias mensagens, o nodo que transmite é

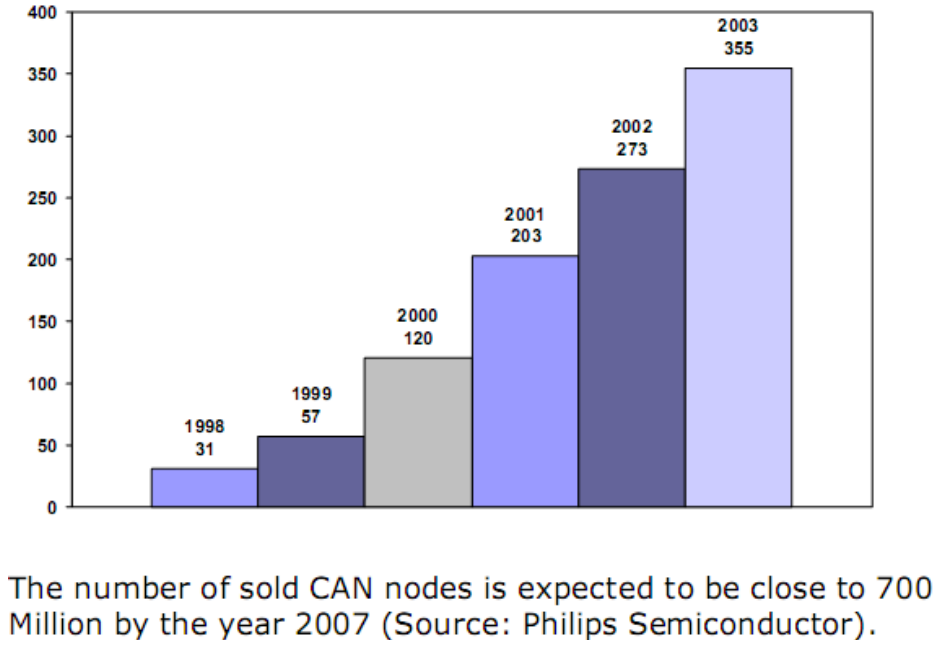


Figura A.2: Número de nodos CAN vendidos.

o que tem a mensagem de menor ID (maior prioridade) ganhando a arbitragem no barramento e continuando a transmitir. Os outros nodos passam ao modo de recepção, tal como mostra na Figura A.4. Os nodos que perdem a arbitragem irão começar uma nova arbitragem assim que o barramento estiver livre para um novo acesso. Logo nenhuma das mensagens é perdida devido à perca da arbitragem no barramento.

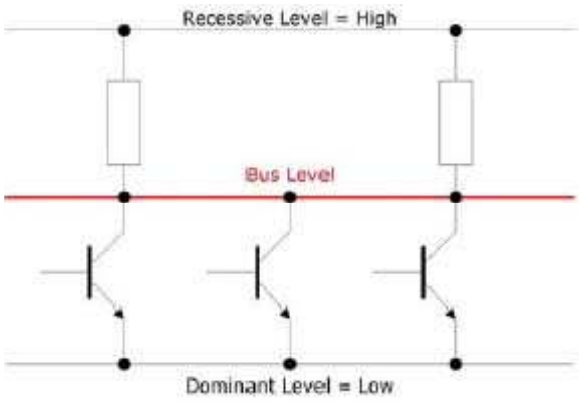


Figura A.3: Implementação da técnica de colector aberto no barramento CAN.

Isto é conseguido pela transmissão do CAN através de um modelo de bits dominantes e recessivos onde o dominante corresponde a um 0 lógico e um recessivo corresponde a um 1 lógico. Isto é conseguido através da implementação física de um colector aberto tal como mostra na Figura A.3. Assim se um nodo transmitir um bit dominante e outro nodo transmitir

um bit recessivo, então o bit dominante ganha.

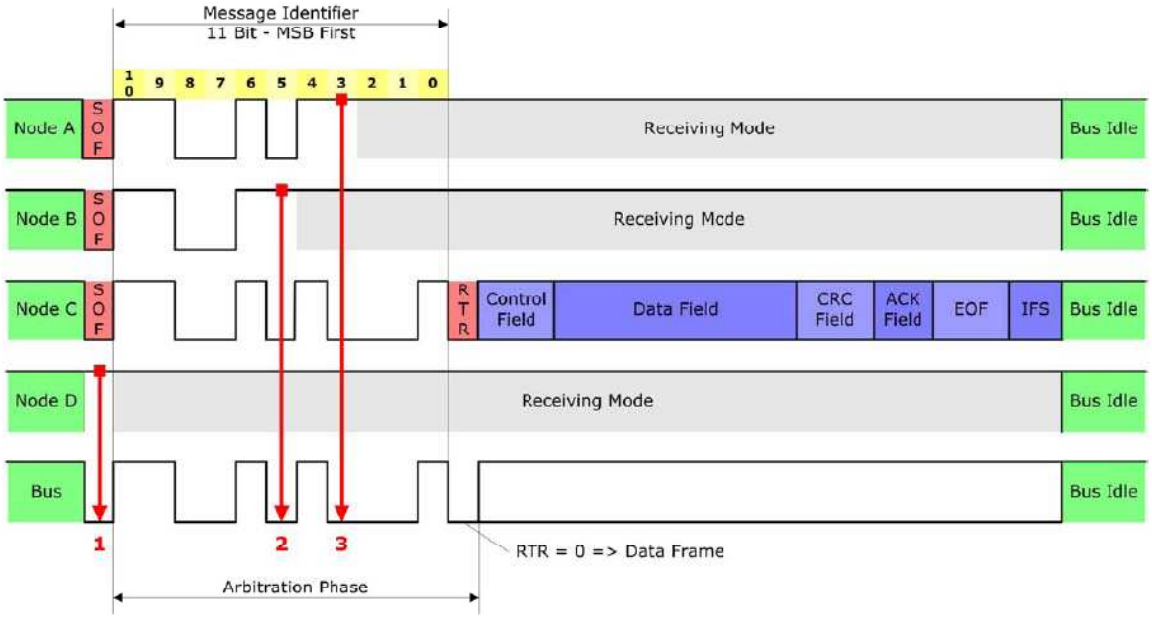


Figura A.4: Arbitragem entre vários nodos CAN.

O CAN usa uma codificação de bit concordante com o princípio *Non-Return-to-Zero* (NRZ). Este fornece a máxima capacidade de transporte, estando o nível do bit constante durante todo o tempo de bit. Este tipo de codificação leva a um insuficiente número de flancos do sinal que são necessários para a sincronização da *Bit stream* (trama de bits) fazendo com que seja necessário aplicar a técnica de *bit stuffing*.

Esta técnica consiste na inserção de um bit complementar (*stuff bit*) na trama depois de 5 bits sucessivos com a mesma polaridade. Este *stuff bit* contribuirá para uma próxima sequência de 5 bits, como é mostrado na Figura A.5 Este bit adicional não irá alterar o conteúdo da mensagem, dado que, o nodo de recepção ao detectar um *stuff bit* na trama retira-o, obtendo-se no final a mensagem original.

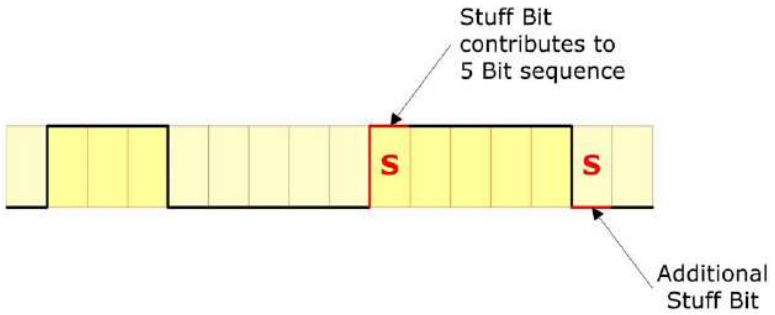


Figura A.5: Exemplo da introdução de *stuff bits* numa trama CAN

A.5.3 Terminologia CAN

A norma CAN prevê a existência de dois formatos para as tramas CAN, são eles a trama *Standard CAN* e a trama *Extended CAN*.

A.5.3.1 Trama *Standard CAN*

Os dispositivos CAN enviam dados através da rede CAN em pacotes chamados tramas. Uma trama *Standard CAN*, tal como mostra a Figura A.6, contém os seguintes campos:

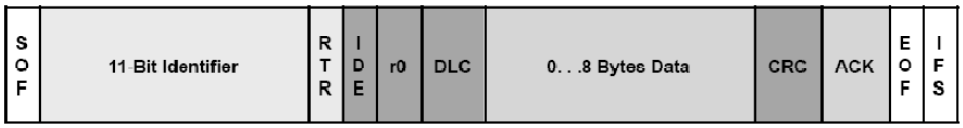


Figura A.6: Campos presentes numa trama *CAN standard*.

SOF (*Start-Of-Frame*) *bit* - Indica o começo de uma mensagem CAN com um bit dominante (0 lógico) e é usado para sincronizar os nodos depois do estado *idle* (desocupado).

Arbitration ID - O formato *standard* CAN com 11 bits de ID identifica a mensagem e indica a prioridade de mensagem. Quanto menor for o número binário mais prioritária será a mensagem.

RTR(*Remote Transmission Request*) *bit* - serve para diferenciar um *remote frame* de um *data frame*. Se o RTR for um bit dominante (zero lógico) indica que é um data *frame*. Caso seja recessivo (um lógico) indica que se trata de um *frame RTR*, ou seja, é um pedido de transmissão.

IDE (*IDentifier Extension*) *bit* - Um bit dominante permite dizer que a mensagem será no formato *standard*.

r0 - bit reservado (para possível alteração da norma).

DLC (*data length code*) - 4 bits que indicam o número de bytes que o campo de dados contém.

Data Field - contém 0-8 bytes de dados.

CRC (*Cyclic Redundancy Check*) - contém 15 bits de código de verificação de redundância cíclica e um delimitador de bit recessivo. O campo CRC é utilizado para a detecção de erros.

ACK (*ACKnowledgement*) *slot* - qualquer controlador CAN que receba correctamente a mensagem envia um bit ACK num fim da mensagem. O nodo de transmissão detecta a presença do bit ACK no barramento e tenta transmitir novamente a mensagem se o bit de ACK não for detectado.

EOF (*End-Of-Frame*) - campo com 7 bits que marca o final da trama CAN e desactiva o *bit-stuffing*, indicando um erro de *stuffing* quando é dominante.

IFS(*Inter-Frame Space*) - Apesar de já não fazer parte da trama CAN representa a quantidade de tempo necessária pelo controlador para mover correctamente a trama recebida para a área de *buffer* de mensagens. Assim sendo, o IFS corresponderá ao tempo mínimo entre dois tramas CAN.

A.5.3.2 *Extended CAN frame*

Como mostra na Figura A.7 a trama *CAN extended* é o mesmo que o *standard* com as seguintes adições:

Figura A.7: Campos presentes numa trama *CAN extended*.

SRR(*Substitute Remote Request*) - Este bit é o substituto do bit RTR da mensagem na norma, que terá um espaço reservado noutro sítio no formato extendido.

IDE (*IDentifier Extension*) *bit* - Um bit recessivo indica que a mensagem será no formato *extended* e que nos bits seguintes virá o resto dos bits do identificador.

r1 - bit adicional reservado.

A.5.4 Aplicações do CAN

O sistema de Barramento série *Controller Area Network* (CAN) foi originalmente desenvolvido para o uso em aplicações automotivas. Hoje a maioria dos construtores de carros de passageiros tem implementado nos seus produtos sistemas que assentam em redes CAN.

Em geral, o CAN é desejável para todas as aplicações onde existam muitos subsistemas baseados em microcontroladores ou em dispositivos que tenham de comunicar. O CAN fornece uma funcionalidade *multi-master* e uma capacidade de tempo-real. Nas características originais do protocolo estão implementados métodos de detecção de erros e confinamento de falhas, que satisfazem as exigências de fiabilidade de equipamento médico, elevadores, e outras aplicações de interacção com o Homem.

Actualmente, os controladores CAN e os *chips* de interface são fisicamente pequenos, estão disponíveis a baixo custo, operam em velocidades de tempo-real e em ambientes bastante agressivos e estão disponíveis no mercado. Todas estas propriedades têm levado a que o CAN também seja usado largamente em outras aplicações além da indústria automóvel.

Por exemplo, sistemas de navegação, sistema de controlo de elevadores, máquinas agrícolas, fotocopiadoras, máquinas de produtos têxteis e até mesmo brinquedos para crianças, usam actualmente soluções baseadas em CAN para as suas redes de comunicação.

A maioria dos carros de passageiros e camiões de marcas europeias como Mercedes-Benz Audi, Renault, Volkswagen,Volvo e outras, têm redes CAN nos seus sistemas de controlo do motor a 500 kbit/s e têm também redes de 125 kbit/s para ligar os vários ECUs (*Electronic Control Unit*). Existe também uma terceira classe, que consiste em redes baseadas em CAN que ligam os vários dispositivos de entretenimento, tal como telemóveis e écrans. Na Figura A.8 é possível ver num carro Volvo todos os tipos de redes CAN referidas anteriormente.

Os aspectos relacionados com segurança do uso de CAN nos carros atraíram a atenção dos construtores de sistemas médicos. Devido à inerente fiabilidade de transmissão de dados e os exigentes requisitos de segurança com que devem ser construídos em sistemas médicos, como máquinas de raio-X (Figura A.9) e sistemas de radioterapia.

As redes CAN são também usadas abundantemente na indústria e geralmente no âmbito de aplicações de controlo. Em particular, as soluções baseadas em CAN estão bem adaptadas

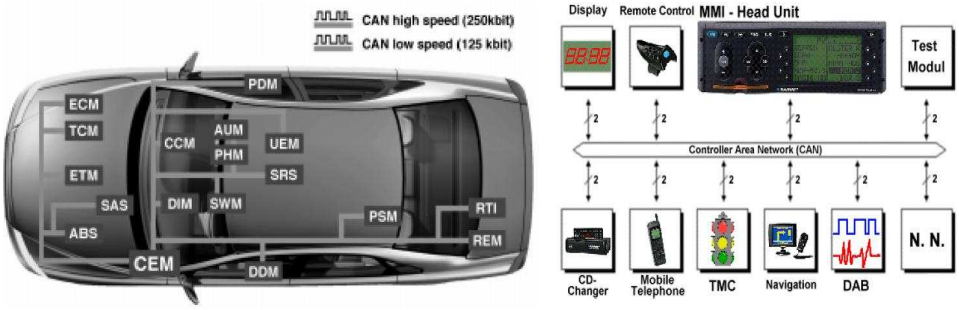


Figura A.8: *Lado esquerdo:* Carro Volvo com as redes CAN de controlo do motores (500 kbit/s) e dos vários ECUs (125 kbit/s)
Lado direito: Dispositivos de entretenimento que normalmente usam redes CAN para como meio de comunicação.

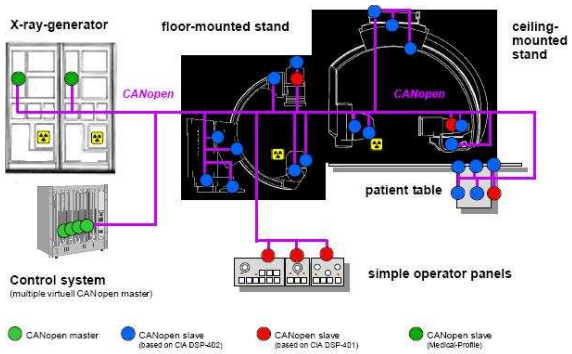


Figura A.9: Exemplo de uma aplicação médica em CAN (máquina de raio-X).

quando é necessário o controlo de movimentos ou outras funções inteligentes, sendo que a maioria dos robots e sistemas de manuseamento usam redes CAN, tal como se pode ver na figura A.10 o exemplo de um robot industrial que faz uso de CAN na comunicação entre os vários dispositivos.

Na indústria, assim como em outras aplicações, torna-se necessário um nível de comunicação mais alto, para tirar partido de funções extras além das que o CAN fornece, tal como, enviar mensagens com mais de 8 bytes e permitir outros modelos de comunicação (*client/server* e *producer/consumer*). Para isso, existem diferentes protocolos *standards* de alto nível disponíveis que tem como espinha dorsal o CAN, tal como, o *CANopen* e o *DeviceNet* e outras.

A.6 Módulo de registo de funções na rotina de interrupção

Os microcontroladores, nomeadamente os da família dos PIC18FXXX, têm dois níveis de interrupção que são atendidos em duas rotinas distintas. Ao adicionar uma nova fonte de interrupção, esta implica acrescentar código dentro da rotina. Esta situação é de evitar dado que as rotinas podem tornar-se muito longas, e o facto de termos que escrever código no meio

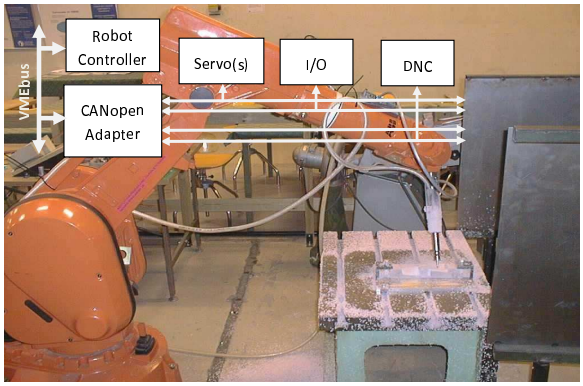


Figura A.10: Robot industrial que utiliza o *CANopen* para a comunicação entre os vários dispositivos.

de outros módulos é propenso de cometer erros.

Assim, por forma ao utilizador se poder abstrair dessa situação foi desenvolvido um módulo que permite invocação de funções na rotina de interrupção através do ponteiro das mesmas. Basicamente o que o utilizador/programador tem de fazer é colocar o código que pretende na rotina de interrupção dentro de uma função e registá-la através do seu ponteiro e escolher o nível em que a pretende registar.

Na imagem seguinte é ilustrado um exemplo muito simples da utilização deste módulo:

```
void main (void)
{

        int_init(); /*limpa os arrays de ponteiros */


        if(add_isr(func_timer0,high_level)!= OK)
                while(1);       /* não executa mais */

        if(add_isr(func_CAN_rx,low_level)!= OK)
                while(1);       /* não executa mais */

}
```

Figura A.11: Exemplo da utilização do módulo de interrupções.

Como se pode ver na Figura A.11 além de incluir o módulo de interrupções no projecto é necessário, primeiro invocar a rotina *int_init* que é responsável por limpar os *arrays* que irão conter os ponteiros das funções. Seguidamente basta apenas registar as funções pretendidas através da rotina *add_isr*, mediante a passagem do ponteiro da função e do nível em que se pretende registar. Esta função devolve a informação se o processo de registo teve sucesso ou insucesso, permitindo assim detectar a introdução de dados inválidos bem como a situação em que o *array* de ponteiros já não tem espaço para registar a função. Os protótipos das funções e os *defines* utilizados neste módulo de interrupções encontram-se nas Figuras A.12 e A.13, respectivamente.

O módulo vem preparado para registar até 6 funções para cada nível de interrupção, no

```
signed char add_isr( void *point_func,short level)

void low_int_init(void)
```

Figura A.12: Protótipos das funções.

entanto este valor é facilmente reconfigurável através da alteração do valor *int_MAX* (Figura A.13).

```
/*defines gerais*/
#define OK              0
#define KO              -1

#define int_MAX         6

#define low_level       0
#define high_level      1
```

Figura A.13: *Defines* gerais utilizados no módulo.

No entanto, nem tudo são vantagens dado que a invocação das funções dentro das rotinas de interrupção levará à introdução de um *overhead* no atendimento das mesmas. Assim e como se pode ver na Figura A.14, quantas mais funções tivermos registadas maior será o *ovearhead* e o atraso no seu atendimento.

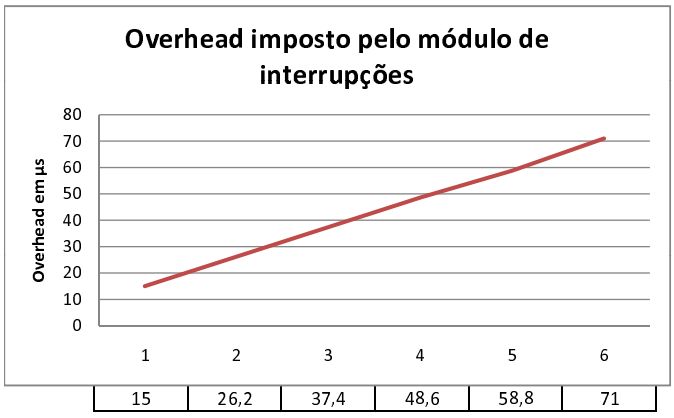


Figura A.14: *Overhead* imposto pelo módulo de interrupções em função do número de funções registadas na rotina ISR.

Da Figura A.14, há a acrescentar que estes resultados são relativos a um PIC a 20MHz, sendo que num PIC ao dobro da frequência era de esperar que os valores fossem metade dos obtidos. Estes tempos representam o tempo despendido a percorrer e validar o *array* bem como o tempo gasto na salvaguarda e reposição de contexto de cada função invocada.

Assim, imaginando uma situação prática em que temos 2 funções registadas em cada nível, sendo as duas de baixa prioridade quem geram em média interrupções de 10 em 10 ms enquanto que as duas de alta prioridade geram interrupções de 8 em 8 ms, num PIC a

funcionar a 20 MHz, representa uma de carga adicional de

$$\frac{2}{10ms} \times 26.2us + \frac{2}{8ms} \times 26.2us = 0.01179 \Rightarrow 1.18\% \quad \text{(A.1)}$$

O valores obtidos são satisfatórios sendo que na maioria das aplicações se revela desprezável o *overhead* introduzido bem como o atraso no atendimento das interrupções. Mas em casos como no protocolo de sincronização, a utilização do módulo provoca uma adicional perda de exactidão na obtenção do *timestamp*, que representa assim uma pior sincronização de relógio. Mas mesmo neste caso, como os valores de atrasos introduzidos são determinísticos, permitem compensar o valor de *timestamp* obtido.

A.7 Medidor de *offset*

O medidor de *offset* permite a partir de dois sinais de entrada calcular o *offset* entre as activações dos mesmos, ou seja, mede o tempo que passou desde a activação de um sinal até activação do outro sinal, tal como se pode ver na Figura A.15.

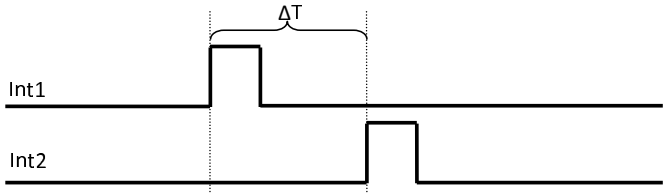


Figura A.15: Tempo medido com a aplicação medidor de *offset*.

O objectivo global desta aplicação é medir em microsegundos o *offset* entre a activação dos sinais e posteriormente enviar esse valor para o PC através de comunicação série. No PC será utilizado o Matlab para armazenar e tratar os dados recebidos. A primeira solução para esta aplicação foi realizada num PIC a 20MHz tirando partido das interrupções externas do mesmo. Mas devido aos tempos gastos nas salvaguardas de contexto e na obtenção dos *timestamps*, eram introduzidos atrasos variáveis que levavam a um erro significativo na medida do *offset*. Os melhores resultados que se conseguiram com esta implementação andavam na ordem dos $4\mu s$, sendo que erros desta grandeza degradariam os resultados obtidos.

Optou-se então por implementar este medidor de *offset* numa FPGA, tirando partido da maior frequência de funcionamento bem como da possibilidade de executar múltiplas tarefas em simultâneo, obtendo assim uma ferramenta de medida robusta e exacta. Na Figura A.16 são ilustrados os grandes blocos desta implementação.

Começando a descrição pelas entradas dos sinais aos quais se pretende medir o *offset*, verifica-se a necessidade de passar estes por um módulo de captura, para que assim possam estar estáveis ao serem analisados pelo bloco *Control*. Este bloco é responsável pela obtenção dos *timestamps* bem como pelo desencadear do envio no bloco *Send control*. Estes *timestamps* resultam da medição do valor do bloco *Timer* que possui um temporizador com incrementos de 1 em $1\mu s$. O bloco *Data format* servirá para efectuar a diferença entre os dois *timestamps* e depois converter o resultado hexadecimal em ASCII para posteriormente serem enviados pelo módulo *UART*. No bloco *Send control* é controlado o envio dos vários *chars* relativos

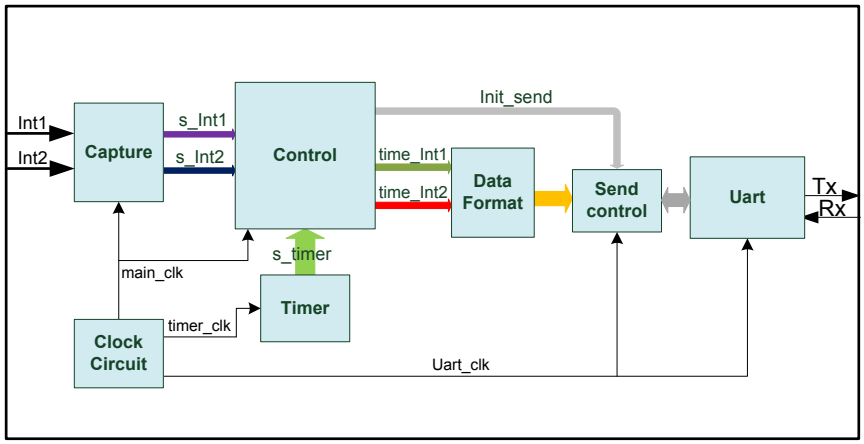


Figura A.16: Arquitectura básica do medidor de *offset*.

a cada captura, seguindo-se do envio do caracter de nova linha ($\backslash n$), sendo a activação do processo de envio desencadeado pelo bloco *Control*.

Foi ainda necessário a construção de um cabo que convertesse as tensões de 0 a +5 volts para as tensões da FPGA, ou seja, de 0 a +3 volts. Além disso, foi necessário ligar o sinal de massa dos PICs ao sinal de massa da FPGA para que assim ambas tivessem a mesma tensão de referência.

A.7.1 Recepção e tratamento de dados - Matlab

Para tornar os resultados enviados do lado da FPGA tratáveis do lado do computador, foi necessário desenvolver um *script* em Matlab para esse efeito. Este procede ao estabelecimento da comunicação série, realizando leituras por linhas, dado que cada resultado se encontra separado por numa linha diferente. Seguidamente este efectua a conversão do valor lido para um valor decimal com sinal guardando-o num *array* para ser posteriormente analisado, continuando neste processo por um número de leituras definido pelo utilizador. Finalmente, com este *array* pode-se obter gráficos e histogramas bem como informação sobre média, desvio padrão, valor máximo e valor mínimo dos valores contidos no *array*.

A.8 Medidor de bloqueio/tempo de execução

O medidor de tempos de bloqueio/tempo de execução surgiu da necessidade de se saber o tempo gasto na rotina de interrupção e de medir o tempo que uma tarefa demora a executar. Para isso basta apenas activar uma *flag* no início da tarefa e desactiva-la imediatamente antes de terminar a tarefa. Assim a partir do tempo activo da *flag* conseguimos obter o tempo de bloqueio da tarefa, tal como se pode ver na Figura A.17.

Esta medida temporal permitirá justificar os resultados conseguidos pelo protocolo de sincronização bem como justificar justificar em parte a gama de *offset* entre um relógio *Master* e um relógio *Slave*. O tempo que o CPU fica bloqueado na rotina de interrupção prejudicará a obtenção do *timestamp* bem como introduzirá bloqueios na aplicação a decorrer no PIC, provando atrasos variáveis na leitura do tempo. Na figura A.18 é ilustrada a arquitectura

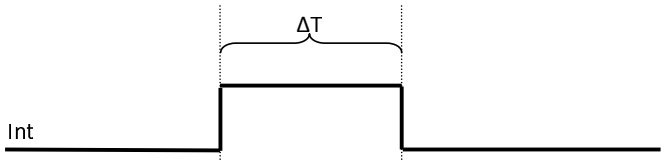


Figura A.17: Tempo medido na implementação medidor de bloqueios.

básica desta aplicação.

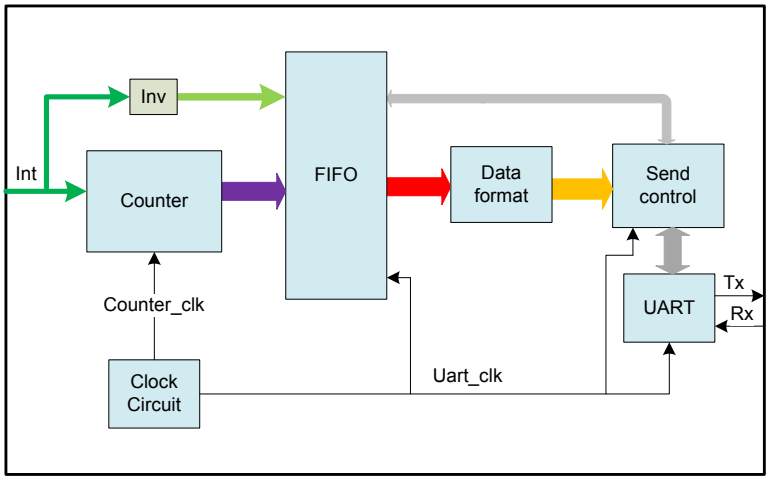


Figura A.18: Arquitectura básica do medidor de bloqueios.

O sinal de entrada *Int* contém o tempo de bloqueio a medir. Este será utilizado de duas formas, primeiro permitirá o incremento do contador no bloco *Counter* quando está no nível lógico 1, depois na transição de 1 para 0 forçará a escrita na FIFO com o valor do contador. Esta dupla funcionalidade pode ser melhor entendida na Figura A.19.

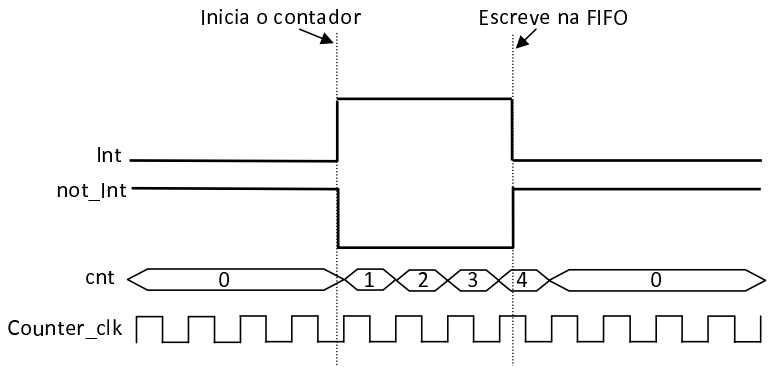


Figura A.19: Diagrama temporal que ilustra a função do sinal *Int*.

Sendo que o *enable* da escrita na FIFO está sempre activo, ao ser invertido o sinal *Int*

no bloco *Inv*, a sua transição de 1 para 0 tornar-se-á numa transição de 0 para 1. Assim ao ser utilizado como relógio de escrita, provocará a escrita do valor do contador na FIFO na transição referida.

Nesta aplicação ao contrário do medidor de *offset*, o tempo entre medições de tempos de bloqueios pode não ser o suficiente para enviar a mensagem anterior pela UART. Assim torna-se necessário utilizar uma memória FIFO para guardar os resultados dos tempos de bloqueios para do lado da UART irem sendo lidos à medida que a esta vai enviando as mensagens.

O bloco *Send control* controla o envio de mensagens pela porta série. Sempre que a FIFO não estiver vazia e já tiver sido enviada a mensagem anterior, este procede a uma nova leitura na FIFO para posteriormente enviar a mensagem pela UART. Finalmente no bloco *Data format*, o valor hexadecimal do tempo de bloqueio lido da FIFO é convertido para código ASCII para poder ser enviado pela UART.

O programa utilizado do lado do computador para recepção e tratamento de dados segue o mesmo princípio referido para o medidor de *offset*.

Bibliografia

[BFRW00] Matthew Bennett, Michael F.Schatz, Heidi Rockwood, and Kurt Wiesenfeld. Huygens clocks. *Royal Society*, 2000.

[BVAK06] Leonidas Bleris, Panagiotis Vouzis, Mark Arnold, and Mayuresh Kothare. A Co-Processor FPGA Platform for the Implementation of Real-Time Model Predictive Control. In *Proceedings of the 2006 American Control Conference*, 2006.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. Distributed Systems: Concepts and Design. pages 211–211, 1994.

[Cor] M. Eng Diarmuid Corry. IEEE1588 - A solution for synchronization of networked data aquisition systems.

[DI] Alexandra Dopplinger and Jim Innis. Aligning System Clocks Over Networks With IEEE1588 Remote Timing Standard.

[DLM02] B. C. Huffman D. L. Mills, A. Thyagarjan. Internet Timekeeping Around the Globe. 2002.

[dO07] Arnaldo Silva Rodrigues de Oliveira. *Especialização e Síntese de Processadores para Aplicação em Sistemas de Tempo-real.* PhD thesis, Universidade de Aveiro, 2007.

[DP06] Chris Dick and Henrik Pedersen. Design and Implementation of High-Performance FPGA Signal Processing Datapaths for Software Defined Radios. 2006.

[eA01] Etschberger et. Al. Controller Area Network, Basics, Protocols, Chips and Applications. *IXXAT Press*, 2001.

[EC98] John Eidson and Wesley Cole. Ethernet rules closed-loop system. pages 39–42, 1998.

[Eida] John Eidson. Recent Advances in IEEE 1588 Technology and Applications. November.

[Eidb] John C. Eidson. Recent Advances in IEEE 1588 Technology and Its Applications. 7.

[Eid05a] John Eidson. Tutorial, IEEE1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. October 2005.

[Eid05b] John C. Eidson. IEEE 1588 Applications in Measurement, Control, and Communication. January 2005.

[Eid05c] John C. Eidson. The Application of IEEE 1588 to Test & Measurement Systems. 2005.

[Eid06] John Eidson. Test & Measurement Application of IEEE 1588. 2006.

[EL] John C. Eidson and Kang Lee. Sharing a Common Sense of Time.

[Esh07] Benjamin D. Esham. Network_Time_Protocol_servers_and_clients.svg. http://en.wikipedia.org/wiki/Network_Time_Protocol, 9 2007.

[fhsc93] Controller Area Network (CAN) for high-speed communication. Road vehicles - Interchange of digital information, 1993.

[Gmb91] Robert Bosch GmbH. *CAN Specification*. BOSH, Postfach 30 02 40, D-70442 Stuttgart, 1991.

[GS94] Martin Gergeleit and Hermann Streich. Implementing a Distributed High resolution Real-Time Clock using the CAN-bus. 1994.

[HGH+02] Roland Höller, Günther Gridling, Martin Horauer, Nikolaus Kerö, Ulrich Schmid, and Klaus Schossmaier. SynUTC High Precision Time Synchronization over Ethernet Networks. In *The SynUTC-project*, 2002.

[HM04] Kendal R. Harris and Anatoly Moldovansky. In sync - New standard allows clocks to link across distributed network, 2004.

[HMF+00] Florian Hartwich, Bernd Müller, Thomas Führer, Robert Hugel, and Robert Bosch GmbH. CANNetwork with Time Triggered Communication. 2000.

[IEE02] IEEE. *IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Published by The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, November 2002.

[JCEW03] Mike Fischer John C. Eidson and Joe White. IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *34th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 243–254, 2003.

[JS05] Ralf Joost and Ralf Salomon. Advantages of FPGA-based multiprocessor systems in industrial applications. In *IEEE 2005*, 2005.

[KCB] Nick Barendt Kendall Correll and Michael Branicky. Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol.

[Kis93] Barbara Kislov. Practical uses of syhchronized clocks in distributed systems. pages 211–211, 1993.

[Kop97] Hermann Kopetz. *Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[LA03] Dongik Lee and Jeff Allan. Fault-Tolerant Clock Synchronisation with Microsecond-Precision for CAN Networked Systems. 2003.

[LE03] Kang Lee and John Eidson. Workshop on IEEE-1588, Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. 2003.

[LGK06] Patrick Loschmidt, Georg Gaderer, and Nikolaus Kerö. IEEE 1588 Hardware for Fault Tolerance and High Precision. 2006.

[LM04] Pedro Leite and Ricardo Marau. RTKPIC18(Real-Time Kernel PIC18FXX8) - Breve Manual , Janeiro 2004.

[McC07] Alex McCarthy. Special Focus: Understanding the IEEE 1588 Precision Time Protocol. http://zone.ni.com/devzone/cda/pub/p/id/130, 2007.

[Mic06] Microchip. PIC18FXXX Data Sheet. Microchip Technology Inc, 2006. 28/40-Pin High-Performance, Enhanced Flash Microcontrollers with CAN Module.

[Mol05] Anatoly Moldovansky. Application of IEEE 1588 in Industrial Automation and Motions Control Systems. October 2005.

[Mul94] Sape Mullender. Distributed Systems. 1994.

[OAF05] Arnaldo S. R. Oliveira, Nelson L. Arqueiro, and Pedro N. Fonseca. CLAN A Technology Independent Synthesizable CAN Controller. In *iCC 2005*, 2005.

[ODV03] ODVA. ODVA Adds Time Synchronization Services for Real-Timeto EtherNetIP and DeviceNet. 2003.

[RGR98] Luís Rodrigues, Mário Guimarães, and José Rufino. Fault-tolerant Clock Synchronization in CAN. 1998.

[RNBP03] Guillermo Rodríguez-Navas, José-Juan Bosh, and Julián Proenza. Hardware Design of a High-precision and Fault-tolerant Clock Subsystem for CAN Networks. 2003.

[RNP03] Guillermo Rodríguez-Navas and Julián Proenza. Clock Synchronization in CAN Distributed Embedded Systems. 2003.

[RVA+98] Jose Rufino, Paulo Verissimo, Guilherme Arroz, Carlos Almeida, and Luis Rodrigues. Fault-Tolerant Broadcasts in CAN. In *Symposium on Fault-Tolerant Computing*, pages 150–159, 1998.

[SBM06] Constantin Siriteanu, Steven Blostein, and James Millar. FPGA-Based Communications Receivers for Smart Antenna Array Embedded Systems. In *EURASIP Journal on Embedded Systems*, 2006.

[Sko01] Paul Skoog. The value of network time synchronization. 2001.

[SSHL99] Klaus Schossmaier, Ulrich Shmid, Martin Horauer, and Dietmar Loy. Specification and Implementation of the Universal Time Coordinated Synchronization Unit. 1999.

[Tan] Alexander E Tan. Technical Article: IEEE 1588 and the need for Ethernet Clock Synchronisation.

[Tan05] Alexander Tan. IEEE 1588 and the need for Ethernet Clock Synchronisation. http://ethernet.industrial-networking.com/articles/articledisplay.asp?id=1943, 2005.

[TBW95] K.W. Tindell, A. Burns, and A. J. Wellings. Calculating Controller Area Network (CAN) message response times. In *Control Engineering Practice*, pages 1163–1169, 1995.

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. Distributed Systems. 2002.

[VR92] Paul Veríssimo and Luís Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. 1992.

[VRC97] Paul Veríssimo, Luís Rodrigues, and António Casimiro. Cesiumpray: a precise and accurate global time service for large-scale systems. pages 243–294, 1997.

[VS04] Dr. André Vallat and Dominik Schneuwly. Clock Synchronization in Telecommunications via PTP (IEEE 1588). In *2004 IEEE 1588 conference*, 2004.

[WB] Hans Weibel and Dominic Béchaz. IEEE 1588 Implementation and Performance of Time Stamping Techniques.

[Wei] Prof. Hans Weibel. High Precision Clock Synchronization according to IEEE 1588 Implementation and Performance Issues.

[ZTC07] Xiang Zhang, Xiao Tang, and Ji Chen. Time synchronization of hierarchical real-time networked CNC system based on ethernet/internet. 2007.